Appendices with PVS source code and proofs for the document

Nova Micro-Hypervisor Verification

B PVS Theory Sources

Contents for Appendix B

B.1 abstract_data.pvs

```
Uninterpreted_Data : Theory
Begin
  Importing IA32

  Uninterpreted_data_type : Type =
    [#
        size : nat,
        valid? : [list[Byte], Address -> bool]
    #]

  uninterpreted_data_type? : PRED[Uninterpreted_data_type] =
    Lambda(uidt : Uninterpreted_data_type) :

      (Forall(l : list[Byte], a : Address) :
        not length(l) = size(uidt) Implies
                valid?(uidt)(l,a) = False)

  % proof status :-)
  u_data_type_length : Lemma
    Forall(uidt : (uninterpreted_data_type?), l : list[Byte], a : Address) :
      valid?(uidt)(l,a) Implies length(l) = size(uidt)

End Uninterpreted_Data


Interpreted_Data[Data : Type] : Theory
Begin

  IMPORTING Uninterpreted_Data, Address_Util, Bitlist_ops

  Interpreted_data_type : Type =
    [#
        uidt : Uninterpreted_data_type, % Assumptions:
        to_byte : [Data, Address -> list[Byte]], % to_byte is zero at unmodified bits
        to_mask : [Data, Address -> list[Byte]], % to_mask is one at unmodified bits
        from_byte : [list[Byte], Address -> lift[Data]]
    #]
  % Thus, new_value = (read & ~to_mask) | to_byte
  % and from_byte must be reasonable on &= to_mask values

  % proof status :-)
  % interpreted_data_type?_TCC1 : TCC Obligation

  interpreted_data_type? : PRED[Interpreted_data_type] =
    Lambda(idt : Interpreted_data_type) :
      uninterpreted_data_type?(idt`uidt) AND
        % result of to_byte is valid (and has length size)
      (Forall(d : Data, a : Address) :
         valid?(uidt(idt))(to_byte(idt)(d,a), a)) And
```

```
        % from_byte fails on invalid stuff
      (Forall(l : list[Byte], a : Address) :
         valid?(uidt(idt))(l, a) IFF
           up?(from_byte(idt)(l, a))) And

        % from_byte is the left inverse of to_byte
        % 1. from_byte does not fail on to_byte results
        % this follows from the previous two
        %
        % (Forall(d : Data, a : Address) :
        % up?(from_byte(idt)(to_byte(idt)(d,a), a))) AND

        % 2. the result is the same
      (Forall(d : Data, a : Address) :
         down(from_byte(idt)(to_byte(idt)(d,a), a)) = d)

pod_data_type? : PRED[Interpreted_data_type] =
  Lambda(idt : Interpreted_data_type) :
    interpreted_data_type?(idt) AND
    (Forall(l : list[Byte], a1, a2 : Address) :
        valid?(uidt(idt))(l, a1) IFF valid?(uidt(idt))(l, a2)) And
     size(uidt(idt)) > 0

% subtype judgements

% the first two are intuetively right, but are not accepted because
% of the subrecord structure

% interpreted_data_is_uninterpreted_data : Judgement
% (interpreted_data_type?) SUBTYPE_OF (uninterpreted_data_type?)

% pod_id_uninterpreted_data : Judgement
% (pod_data_type?) SUBTYPE_OF (uninterpreted_data_type?)

% proof status :-)
pod_is_interpreted_data : Judgement
  (pod_data_type?) SUBTYPE_OF (interpreted_data_type?)

% proof status :-)
pod_size : Lemma
  Forall (dt : (pod_data_type?)) : size(uidt(dt)) > 0

% Utility predicates

idt : Var (interpreted_data_type?)
addr : Var Address
min : Var Memory_Address
max : Var Memory_Address

in_memory?(idt, addr, min, max) : bool =
  in_memory(min, max)(addr) And in_memory(min, max)(addr + size(uidt(idt)))
```

```
  in_blessed_memory?(idt, addr, (address_range : PRED[Address])) : bool =
    subset?(address_block(addr, size(uidt(idt))), address_range)

  % proof status :-)
  data_type_length : Lemma Forall(l : list[Byte]) :
      valid?(uidt(idt))(l,addr) Implies length(l) = size(uidt(idt))

  % proof status :-)
  length_to_byte : Lemma Forall(data : Data) :
    length(to_byte(idt)(data, addr)) = size(uidt(idt))

  % proof status :-)
  valid_from_byte : Lemma Forall(l : list[Byte]) :
    idt`uidt`valid?(l, addr) Implies up?(from_byte(idt)(l, addr))

  % proof status :-)
  valid_iff_from_byte : Lemma Forall(l : list[Byte]) :
    idt`uidt`valid?(l, addr) <=> up?(from_byte(idt)(l, addr))

  % proof status :-)
  valid_to_byte : Lemma Forall(data : Data) :
    idt`uidt`valid?(to_byte(idt)(data, addr), addr)

  % proof status :-)
  up_from_byte_to_byte : Lemma Forall(data : Data) :
    up?(from_byte(idt)(to_byte(idt)(data, addr), addr))

  % proof status :-)
  from_byte_to_byte : Lemma Forall(data : Data) :
    from_byte(idt)(to_byte(idt)(data, addr), addr) = up(data)

  % proof status :-)
  in_blessed_memory_subset : Lemma
    Forall (address_range : PRED[Address]) :
    in_blessed_memory?(idt, addr, address_range) Implies
      subset?(address_block(addr, size(uidt(idt))), address_range)

End Interpreted_Data


Interpreted_Data_Lift[Data1, Data2 : Type] : Theory
Begin

  Importing Interpreted_Data

  dt_lift(dt : Interpreted_data_type[Data1],
          ext : [Data2 -> Data1],
          inj : [Data1 -> Data2]) : Interpreted_data_type[Data2] =
    (# uidt := uidt(dt),
       to_byte := Lambda (d2 : Data2, a : Address) : to_byte(dt)(ext(d2), a),
       to_mask := Lambda (d2 : Data2, a : Address) : to_mask(dt)(ext(d2), a),
```

```
    from_byte := Lambda (l : list[Byte], a : Address) :
                   Cases from_byte(dt)(l, a) Of
                     up(d1) : up(inj(d1)),
                     bottom : bottom
                   EndCases
  #)

End Interpreted_Data_Lift


Abstract_Read_Write[State, Data : Type] : Theory
Begin
  IMPORTING Memory_Change[State]
  IMPORTING State_Transformer_Lift, Interpreted_Data

  dt : Var (interpreted_data_type?[Data]) % dt aka Data_type
  addr : Var Address
  s : Var State % s aka state
  data : Var Data
  pm : Var Memory_struct[State]

  valid_in_mem(pm,dt)(addr)(s) : bool =
    Cases memory_read_list(pm)(addr, size(uidt(dt)))(s) OF
      OK(sn, bl) : dt`uidt`valid?(bl, addr)
      Else true
    EndCases

  read_data(pm,dt)(addr) : [State -> ExprResult[State, Data]] =
    (memory_read_list(pm)(addr, size(uidt(dt))) ##
       Lambda(bl : list[Byte]) : ok_lift(from_byte(dt)(bl, addr)))

  write_data(pm,dt)(addr, data) : [State -> ExprResult[State, Unit]] =
    memory_write_list(pm)(addr, to_byte(dt)(data, addr))

  % proof status :-)
  read_data_valid_in_mem : Lemma
    OK?(read_data(pm, dt)(addr)(s)) Implies valid_in_mem(pm, dt)(addr)(s)

End Abstract_Read_Write
```

B.2 allocators.pvs

```
Allocator[State : Type] : Theory
% Specification of memory allocators working on plain memory,
% such as the stack allocator and the one for global variables.
Begin
  Importing Plain_Memory[State]

  % The allocator interface consists of two real methods and four
  % logical ones.
  % alloc allocates memory
```

```
  % free frees memory
  %
  % memory_pool The memory from which the allocator allocates. This should
  % be disjoint from all other live allocators.
  % allocated Gives the set of addresses that have been allocated by this
  % allocator. Used to get the disjointness of correctly
  % allocated variables.
  % private_mem Returns those memory regions that must not be changed
  % for this allocator to work properly.
  % freed_size Returns the size of the block that will be freed by
  % a free in the same state. Needed to require a correct
  % adjustment of the allocated predicate during free.

  % We exclude the possibility to allocate 0 Bytes here (this is
  % permitted by many malloc libraries). We further require that free
  % makes a visible change in the allocated predicate. Some malloc
  % libraries (or was it the C++ standard?) do not guarantee that freed
  % memory is really freed. We only require that the predicate is changed
  % (and not that freed memory can be allocated again), nevertheless, this
  % excludes empty free functions (such as in conservative garbage
  % collectors).
  %
  % One would expect a valid predicate, which is turned on by init and
  % maintained afterwards. This valid predicate would somehow classify
  % the set of states in which the allocator is expected to work.
  % However, the real methods alloc and free
  % have always the possibility to return fatal. The valid predicate would
  % only clutter the specification, therefore we keep it implicit.
  % The set of states in which the allocator is expected to work is
  % given by the states predicate of the involved plain memory.

  Allocator : Type = [#
    memory_pool : [(plain_memory?) -> [State -> PRED[Address]]],
    allocated : [(plain_memory?) -> [State -> PRED[Address]]],
    private_mem : [(plain_memory?) -> PRED[Address]],
    freed_size : [(plain_memory?) -> [Address -> [State -> lift[posnat]]]],

    % init: [(plain_memory?) -> [State -> ExprResult[State, Unit]]],
    alloc: [(plain_memory?) ->
                          [posnat -> [State -> ExprResult[State, Address]]]],
    free: [(plain_memory?) -> [Address -> [State -> ExprResult[State, Unit]]]]
  #]


  % proof status :-)
  % null_address_TCC1 : TCC Obligation

% HT: This should be NULL for an arbitrary pointer type, coming from
% the pointer semantics
  null_address : Memory_Address = Mem(0)

  % Required properties of allocators:
```

```
% − allocated is a subset of the memory pool
% − Newly allocated memory is disjoint and comes from the memory pool.
% − If allocate and free succeed the allocated predicate is adjusted
% accordingly, while the memory pool stays constant.
% − Free succeeds precisely when freed_size does.
% − Reading memory and writing memory at non−private addresses
% does not change the allocated predicate and the memory pool

% proof status :−)
% allocator?_TCC1 : TCC Obligation

% proof status :−)
% allocator?_TCC2 : TCC Obligation

% proof status :−)
% allocator?_TCC3 : TCC Obligation

% proof status :−)
% allocator?_TCC4 : TCC Obligation

% proof status :−)
% allocator?_TCC5 : TCC Obligation

% proof status :−)
% allocator?_TCC6 : TCC Obligation

% proof status :−)
% allocator?_TCC7 : TCC Obligation

% proof status :−)
% allocator?_TCC8 : TCC Obligation

allocator?(pm : (plain_memory?)) : PRED[Allocator] = Lambda(ac : Allocator) :

  % private memory is in plain memory −− therefore the
  % allocator won't change under read access
  subset?(ac`private_mem(pm), union(pm`ro_addr, pm`rw_addr))
  And

  Forall(s : State) : pm`states(s) Implies
    % allocate from the memory pool
    subset?(ac`allocated(pm)(s), ac`memory_pool(pm)(s)) And

    % alloc properties
    (Forall(size : posnat) :
      OK?(ac`alloc(pm)(size)(s)) And
      Not data(ac`alloc(pm)(size)(s)) = null_address Implies
        disjoint?(ac`allocated(pm)(s),
              address_block(data(ac`alloc(pm)(size)(s)), size))
        And
        ac`allocated(pm)(state(ac`alloc(pm)(size)(s))) =
          union(ac`allocated(pm)(s),
```

```
            address_block(data(ac`alloc(pm)(size)(s)), size))
      And
      ac`memory_pool(pm)(state(ac`alloc(pm)(size)(s))) =
        ac`memory_pool(pm)(s))
And

% connection of free <--> freed_size
(Forall(addr : Address) :
  OK?(ac`free(pm)(addr)(s)) IFF up?(ac`freed_size(pm)(addr)(s)))
And

% free properties
(Forall(addr : Address) :
  OK?(ac`free(pm)(addr)(s)) Implies
    ac`allocated(pm)(state(ac`free(pm)(addr)(s))) =
      difference(ac`allocated(pm)(s),
            address_block(addr, down(ac`freed_size(pm)(addr)(s))))
    And
    ac`memory_pool(pm)(state(ac`free(pm)(addr)(s))) =
      ac`memory_pool(pm)(s))
And

% The private memory contains the state of the allocator. If
% it stays constant, then the allocator is unchanged.
(Forall(s2 : State) :
  pm`states(s2) And
  stays_unchanged(pm)(s, s2, ac`private_mem(pm))
    Implies
      ac`memory_pool(pm)(s) = ac`memory_pool(pm)(s2) And
      ac`allocated(pm)(s) = ac`allocated(pm)(s2))
And

% next_state invariants
%(has_next_state(ac`init(pm)(s)) Implies pm`states(state(ac`init(pm)(s))))
%And
(Forall(size : posnat) :
  has_next_state(ac`alloc(pm)(size)(s)) Implies
    pm`states(state(ac`alloc(pm)(size)(s))))
And
(Forall(addr : Address) :
  has_next_state(ac`free(pm)(addr)(s)) Implies
    pm`states(state(ac`free(pm)(addr)(s))))
And

% alloc and free only change private memory, more precisely,
% only writable private memory
(Forall(size : posnat) :
  OK?(ac`alloc(pm)(size)(s)) And
  NOT data(ac`alloc(pm)(size)(s)) = null_address
    Implies
      only_changes(pm)(s, state(ac`alloc(pm)(size)(s)),
                                          ac`private_mem(pm)))
```

```
    And
    (Forall(free_addr : Address) :
      OK?(ac`free(pm)(free_addr)(s)) Implies
        only_changes(pm)(s, state(ac`free(pm)(free_addr)(s)),
                                                ac`private_mem(pm)))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% private memory is inside plain memory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocator_subset_private_memory_plain_memory : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm))) :
    subset?(ac`private_mem(pm), union(pm`ro_addr, pm`rw_addr))

% proof status :-)
allocator_private_memory_in_plain_memory : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)), addr : Address) :
    ac`private_mem(pm)(addr) Implies
      union(pm`ro_addr, pm`rw_addr)(addr)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% allocate from the memory pool
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocator_subset_allocated_memory_pool : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)), s : State) :
    pm`states(s) Implies
      subset?(ac`allocated(pm)(s), ac`memory_pool(pm)(s))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% alloc properties
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocator_alloc_disjoint : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                size : posnat, s : State) :
    pm`states(s) And
    OK?(ac`alloc(pm)(size)(s)) And
    Not data(ac`alloc(pm)(size)(s)) = null_address Implies
      disjoint?(ac`allocated(pm)(s),
```

```
            address_block(data(ac`alloc(pm)(size)(s)), size))

  % proof status :-)
  % allocator_allocated_alloc_TCC1 : TCC Obligation

  % proof status :-)
  allocator_allocated_alloc : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                  size : posnat, s : State) :
      pm`states(s) And
      OK?(ac`alloc(pm)(size)(s)) And
      Not data(ac`alloc(pm)(size)(s)) = null_address Implies
        ac`allocated(pm)(state(ac`alloc(pm)(size)(s))) =
          union(ac`allocated(pm)(s),
                address_block(data(ac`alloc(pm)(size)(s)), size))

  % proof status :-)
  allocator_memory_pool_alloc : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                  size : posnat, s : State) :
      pm`states(s) And
      OK?(ac`alloc(pm)(size)(s)) And
      Not data(ac`alloc(pm)(size)(s)) = null_address Implies
        ac`memory_pool(pm)(state(ac`alloc(pm)(size)(s))) =
          ac`memory_pool(pm)(s)

  % proof status :-)
  allocator_subset_address_block_alloc_allocated : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                size : posnat, addr : Address, s : State) :
      pm`states(s) And
      OK?(ac`alloc(pm)(size)(s)) And
      NOT data(ac`alloc(pm)(size)(s)) = null_address Implies
        subset?(address_block(data(ac`alloc(pm)(size)(s)), size),
                ac`allocated(pm)(state(ac`alloc(pm)(size)(s))))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % connection of free <--> freed_size
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  allocator_free_freed_size : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                          addr : Address, s : State) :
      pm`states(s) Implies
        (OK?(ac`free(pm)(addr)(s)) IFF up?(ac`freed_size(pm)(addr)(s)))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%
% free properties
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_allocated_free_TCC1 : TCC Obligation

% proof status :-)
% allocator_allocated_free_TCC2 : TCC Obligation

% proof status :-)
allocator_allocated_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                addr : Address, s : State) :
    pm`states(s) And
    OK?(ac`free(pm)(addr)(s)) Implies
      ac`allocated(pm)(state(ac`free(pm)(addr)(s))) =
        difference(ac`allocated(pm)(s),
            address_block(addr, down(ac`freed_size(pm)(addr)(s))))

% proof status :-)
allocator_memory_pool_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                addr : Address, s : State) :
    pm`states(s) And
    OK?(ac`free(pm)(addr)(s)) Implies
      ac`memory_pool(pm)(state(ac`free(pm)(addr)(s))) =
        ac`memory_pool(pm)(s)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% constant private memory implies memory_pool and allocated are constant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
allocator_const_private_memory_memory_pool : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)), s1, s2 : State) :
    pm`states(s1) And
    pm`states(s2) And
    stays_unchanged(pm)(s1, s2, ac`private_mem(pm))
      Implies
        ac`memory_pool(pm)(s1) = ac`memory_pool(pm)(s2)

% proof status :-)
allocator_const_private_memory_allocated : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)), s1, s2 : State) :
    pm`states(s1) And
    pm`states(s2) And
    stays_unchanged(pm)(s1, s2, ac`private_mem(pm))
```

```
      Implies
        ac‘allocated(pm)(s1) = ac‘allocated(pm)(s2)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% next_state invariants
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_next_state_alloc_TCC1 : TCC Obligation

% proof status :-)
allocator_next_state_alloc : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                 size : posnat, s : State) :
    pm‘states(s) And
    has_next_state(ac‘alloc(pm)(size)(s)) Implies
      pm‘states(state(ac‘alloc(pm)(size)(s)))

% proof status :-)
% allocator_next_state_free_TCC1 : TCC Obligation

% proof status :-)
allocator_next_state_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                                 addr : Address, s : State) :
    pm‘states(s) And
    has_next_state(ac‘free(pm)(addr)(s)) Implies
      pm‘states(state(ac‘free(pm)(addr)(s)))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% alloc and free only change private memory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_only_changes_alloc_TCC1 : TCC Obligation

% proof status :-)
allocator_only_changes_alloc : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                          size : posnat, s : State) :
    pm‘states(s) And
    OK?(ac‘alloc(pm)(size)(s)) And
    NOT data(ac‘alloc(pm)(size)(s)) = null_address
      Implies
        only_changes(pm)(s, state(ac‘alloc(pm)(size)(s)),
                                                ac‘private_mem(pm))
```

```
% proof status :-)
% allocator_only_changes_free_TCC1 : TCC Obligation

% proof status :-)
allocator_only_changes_free : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                        free_addr : Address, s : State) :
    pm`states(s) And
    OK?(ac`free(pm)(free_addr)(s)) Implies
      only_changes(pm)(s, state(ac`free(pm)(free_addr)(s)),
                                              ac`private_mem(pm))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_allocated_memory_read_TCC1 : TCC Obligation

% proof status :-)
allocator_allocated_memory_read : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                              addr : Address, s : State) :
    pm`states(s) And
    union(pm`ro_addr, pm`rw_addr)(addr) Implies
      ac`allocated(pm)(state(memory_read(pm`mem)(addr)(s))) =
            ac`allocated(pm)(s)

% proof status :-)
allocator_memory_pool_memory_read : Lemma
  Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                              addr : Address, s : State) :
    pm`states(s) And
    union(pm`ro_addr, pm`rw_addr)(addr) Implies
        ac`memory_pool(pm)(state(memory_read(pm`mem)(addr)(s))) =
                ac`memory_pool(pm)(s)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_write invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocator_stays_unchanged_memory_write_private_mem_TCC1 : TCC Obligation

% proof status :-)
% allocator_stays_unchanged_memory_write_private_mem_TCC2 : TCC Obligation
```

```
  % proof status :-)
  allocator_stays_unchanged_memory_write_private_mem : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                   addr : Address, byte : Byte, s : State) :
      pm`states(s) And
      pm`rw_addr(addr) And
      Not ac`private_mem(pm)(addr)
        Implies
          stays_unchanged(pm)(s, state(memory_write(pm`mem)(addr, byte)(s)),
                                        ac`private_mem(pm))

  % proof status :-)
  allocator_allocated_memory_write : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                   addr : Address, byte : Byte, s : State) :
      pm`states(s) And
      pm`rw_addr(addr) And
      Not ac`private_mem(pm)(addr) Implies
        ac`allocated(pm)(state(memory_write(pm`mem)(addr, byte)(s))) =
              ac`allocated(pm)(s)

  % proof status :-)
  allocator_memory_pool_memory_write : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                   addr : Address, byte : Byte, s : State) :
      pm`states(s) And
      pm`rw_addr(addr) And
      Not ac`private_mem(pm)(addr) Implies
        ac`memory_pool(pm)(state(memory_write(pm`mem)(addr, byte)(s))) =
                  ac`memory_pool(pm)(s)

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % more complex properties
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


  % proof status :-)
  allocator_subset_address_block_alloc_memory_pool : Lemma
    Forall(pm : (plain_memory?), ac : (allocator?(pm)),
                                   size : posnat, addr : Address, s : State) :
      pm`states(s) And
      OK?(ac`alloc(pm)(size)(s)) And
      NOT data(ac`alloc(pm)(size)(s)) = null_address Implies
        subset?(address_block(data(ac`alloc(pm)(size)(s)), size),
                        ac`memory_pool(pm)(s))


End Allocator
```

```
Allocation_Info[State : Type] : Theory
% This theory formalizes the (additional) information that
% we track for allocators and its consistency condition.
Begin
  Importing Allocator[State], More_Relations

  % The allocation info contains the set of allocation points of one
  % specific allocator. An allocation point is an (address, size) pair
  % that has been allocated with this very allocator (that is, the
  % address was returned by an alloc with argument size in some former
  % state and no free has been done so far.
  %
  % Note that the consistency below requires only that the allocation
  % info contains a subset of the allocated memory. In the following
  % cases correctly allocated memory must not be in the allocation info:
  %
  % - Memory allocated in allocator A that is used as memory pool of
  % allocator B.

  Allocator_Info : Type = list[[Address, posnat]]

  allocator_info? : PRED[Allocator_Info] = Lambda(ai : Allocator_Info) :
    blocks_pairwise_disjoint(ai)

  pm : Var (plain_memory?)
  ai : Var Allocator_Info
  s : Var State

  consistent_allocator?(pm)(ac : (allocator?(pm)), ai)(s) : bool =
    allocator_info?(ai) And
    Forall(addr : Address, size : posnat) : member((addr, size), ai) Implies
      subset?(address_block(addr, size), ac`allocated(pm)(s))

  % proof status :-)
  consistent_allocator_state_change : Lemma
    Forall(ac : (allocator?(pm)), s1, s2 : State) :
      consistent_allocator?(pm)(ac, ai)(s1) And
      ac`allocated(pm)(s1) = ac`allocated(pm)(s2) Implies
        consistent_allocator?(pm)(ac, ai)(s2)


  % proof status :-)
  % consistent_allocator_read_TCC1 : TCC Obligation

  % proof status :-)
  consistent_allocator_read : Lemma
    Forall(ac : (allocator?(pm)), addr : Address) :
      pm`states(s) And
      consistent_allocator?(pm)(ac, ai)(s) And
      union(pm`ro_addr, pm`rw_addr)(addr) Implies
        consistent_allocator?(pm)(ac, ai)(state(memory_read(pm`mem)(addr)(s)))
```

```
  % proof status :-)
  % consistent_allocator_write_TCC1 : TCC Obligation

  % proof status :-)
  consistent_allocator_write : Lemma
    Forall(ac : (allocator?(pm)), addr : Address, byte : Byte) :
      pm`states(s) And
      consistent_allocator?(pm)(ac, ai)(s) And
      pm`rw_addr(addr) And
      Not ac`private_mem(pm)(addr) Implies
        consistent_allocator?(pm)(ac, ai)
                              (state(memory_write(pm`mem)(addr, byte)(s)))

  % proof status :-)
  % consistent_allocator_alloc_TCC1 : TCC Obligation

  % proof status :-)
  consistent_allocator_alloc : Lemma
    Forall(ac : (allocator?(pm)), size : posnat) :
      pm`states(s) And
      consistent_allocator?(pm)(ac, ai)(s) And
      OK?(ac`alloc(pm)(size)(s)) And
      Not data(ac`alloc(pm)(size)(s)) = null_address Implies
        consistent_allocator?(pm)
          (ac, cons((data(ac`alloc(pm)(size)(s)), size), ai))
          (state(ac`alloc(pm)(size)(s)))

  % proof status :-)
  % consistent_allocator_free_TCC1 : TCC Obligation

  % proof status :-)
  % consistent_allocator_free_TCC2 : TCC Obligation

  % proof status :-)
  consistent_allocator_free : Lemma
    Forall(ac : (allocator?(pm)), addr : Address) :
      pm`states(s) And
      consistent_allocator?(pm)(ac, ai)(s) And
      OK?(ac`free(pm)(addr)(s)) And
      (member((addr, down(ac`freed_size(pm)(addr)(s))), ai) OR
         blocks_pairwise_disjoint(
           cons((addr, down(ac`freed_size(pm)(addr)(s))), ai)))
        Implies
          consistent_allocator?(pm)
            (ac, list_remove((addr, down(ac`freed_size(pm)(addr)(s))), ai))
            (state(ac`free(pm)(addr)(s)))

End Allocation_Info


Allocation_Table[State : Type] : Theory
```

```
Begin
  Importing Allocation_Info[State], Graph, Finite_Set_Reduce,
        More_List_Props_2

  pm : Var (plain_memory?)

  % An allocation table contains a hierarchy of allocators together
  % with their allocation info's. The hierarchy is a finite, directed graph.
  % hierarchy(ac1, ac2) means that the memory pool of ac2 is allocated
  % inside ac1. Currently the hierarchy must be a tree, that is a
  % non-root allocator gets its memory from preciesely one allocator.
  % Thus Fiascos reagion manager is currently unsupported.
  Allocation_Table(pm) : Type = [#
    hierarchy : (finite_tree?[(allocator?(pm))]),
    info : [(hierarchy`nodes) -> Allocator_Info]
  #]

  ai : Var Allocator_Info
  s : Var State

  allocation_ok?(pm)(at : Allocation_Table(pm), s) : bool =
    (Forall(ac1, ac2 : (at`hierarchy`nodes)) :
       at`hierarchy`edges(ac1, ac2) Implies
         subset?(ac2`memory_pool(pm)(s), ac1`allocated(pm)(s)) And
         block_is_free(ac2`memory_pool(pm)(s), at`info(ac1))
    )
    And
    (Forall(ac1, ac2, ac3 : (at`hierarchy`nodes)) :
       at`hierarchy`edges(ac1, ac2) And
       at`hierarchy`edges(ac1, ac3) Implies
         ac2 = ac3 OR
         disjoint?(ac2`memory_pool(pm)(s), ac3`memory_pool(pm)(s))
    )
    And
    (Forall(ac1, ac2 : (roots(at`hierarchy))) :
      ac1 = ac2 OR disjoint?(ac1`memory_pool(pm)(s), ac2`memory_pool(pm)(s)))
    And
    (Forall(ac1, ac2 : (at`hierarchy`nodes)) :
      ac1 = ac2 OR disjoint?(ac1`private_mem(pm), ac2`private_mem(pm)))
    And
    (Forall(ac : (at`hierarchy`nodes)) :
       consistent_allocator?(pm)(ac, at`info(ac))(s))

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % allocation access lemmas
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  allocation_finite_tree : Lemma
    Forall(at : Allocation_Table(pm)) :
```

```
        finite_tree?(at`hierarchy)

  % proof status :-)
  allocation_subnodes_allocated_and_free : Lemma
    Forall(at : Allocation_Table(pm), ac1, ac2 : (at`hierarchy`nodes)) :
      allocation_ok?(pm)(at, s) And
      at`hierarchy`edges(ac1, ac2) Implies
        subset?(ac2`memory_pool(pm)(s), ac1`allocated(pm)(s)) And
        block_is_free(ac2`memory_pool(pm)(s), at`info(ac1))

  % proof status :-)
  allocation_subnodes_disjoint : Lemma
    Forall(at : Allocation_Table(pm), ac1, ac2, ac3 : (at`hierarchy`nodes)) :
      allocation_ok?(pm)(at, s) And
      at`hierarchy`edges(ac1, ac2) And
      at`hierarchy`edges(ac1, ac3) Implies
        ac2 = ac3 OR
        disjoint?(ac2`memory_pool(pm)(s), ac3`memory_pool(pm)(s))

  % proof status :-)
  allocation_roots_disjoint : Lemma
    Forall(at : Allocation_Table(pm), ac1, ac2 : (roots(at`hierarchy))) :
      allocation_ok?(pm)(at, s) Implies
        ac1 = ac2 OR
        disjoint?(ac1`memory_pool(pm)(s), ac2`memory_pool(pm)(s))

  % proof status :-)
  allocation_private_memory_disjoint : Lemma
    Forall(at : Allocation_Table(pm), ac1, ac2 : (at`hierarchy`nodes)) :
      allocation_ok?(pm)(at, s) Implies
        ac1 = ac2 OR
        disjoint?(ac1`private_mem(pm), ac2`private_mem(pm))

  % proof status :-)
  allocation_consistent_allocator : Lemma
    Forall(at : Allocation_Table(pm), ac : (at`hierarchy`nodes)) :
      allocation_ok?(pm)(at, s) Implies
        consistent_allocator?(pm)(ac, at`info(ac))(s)

  % proof status :-)
  allocation_allocated_allocation_point : Lemma
    Forall(at: Allocation_Table(pm), ac : (at`hierarchy`nodes),
                                              addr : Address, size : posnat) :
      allocation_ok?(pm)(at, s) And
      member((addr, size), at`info(ac)) Implies
        subset?(address_block(addr, size), ac`allocated(pm)(s))

  % proof status :-)
  allocation_allocation_point_in_memory_pool : Lemma
    Forall(at: Allocation_Table(pm), ac : (at`hierarchy`nodes),
                                              addr : Address, size : posnat) :
      pm`states(s) And
```

```
    allocation_ok?(pm)(at, s) And
    member((addr, size), at`info(ac)) Implies
      subset?(address_block(addr, size), ac`memory_pool(pm)(s))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% general utility lemmas
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% proof status :-)
allocators_different_in_hierachy : Lemma
  Forall(at : Allocation_Table(pm), ac1, ac2 : (at`hierarchy`nodes)) :
    at`hierarchy`edges(ac1, ac2) Implies
      Not ac1 = ac2

% proof status :-)
uppath_memory_pool : Lemma
  Forall(at : Allocation_Table(pm), p : (path?(at`hierarchy))) :
    allocation_ok?(pm)(at, s) And
    pm`states(s) Implies
      subset?(path_end(p)`memory_pool(pm)(s),
                                  path_start(p)`memory_pool(pm)(s))

% proof status :-)
% uppath_memory_pool_allocated_and_free_TCC1 : TCC Obligation

% proof status :-)
uppath_memory_pool_allocated_and_free : Lemma
  Forall(at : Allocation_Table(pm), p : (path?(at`hierarchy))) :
    pm`states(s) And
    allocation_ok?(pm)(at, s) And
    length(p) >= 2 Implies
      subset?(path_end(p)`memory_pool(pm)(s), path_start(p)`allocated(pm)(s))
      And
      block_is_free(path_end(p)`memory_pool(pm)(s), at`info(path_start(p)))

% proof status :-)
allocation_memory_pools : Lemma
  Forall(at : Allocation_Table(pm), ac1, ac2 : (at`hierarchy`nodes)) :
    pm`states(s) And
    allocation_ok?(pm)(at, s) Implies
      ac1 = ac2
      OR
      disjoint?(ac1`memory_pool(pm)(s), ac2`memory_pool(pm)(s))
      OR
      (subset?(ac1`memory_pool(pm)(s), ac2`allocated(pm)(s)) And
          block_is_free(ac1`memory_pool(pm)(s), at`info(ac2)))
      OR
      (subset?(ac2`memory_pool(pm)(s), ac1`allocated(pm)(s)) And
```

```
        block_is_free(ac2`memory_pool(pm)(s), at`info(ac1)))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% general allocation results
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% allocation_info_TCC1 : TCC Obligation

allocation_info(pm)(at : Allocation_Table(pm)) : Allocator_Info =
  flatten(map(at`info)(list_of_finite_set(at`hierarchy`nodes)))

% proof status :-)
member_allocation_info : Lemma
  Forall(at : Allocation_Table(pm), addr : Address, size : posnat) :
    member((addr, size), allocation_info(pm)(at)) Implies
      Exists(ac : (at`hierarchy`nodes)) : member((addr, size), at`info(ac))

% all allocation points are disjoint
% proof status :-)
allocation_consistent : Lemma
  Forall(at : Allocation_Table(pm)) :
    pm`states(s) And
    allocation_ok?(pm)(at, s) Implies
      allocator_info?(allocation_info(pm)(at))

% proof status :-)
% allocation_alloc_other_private_memory_TCC1 : TCC Obligation

% proof status :-)
% allocation_alloc_other_private_memory_TCC2 : TCC Obligation

% proof status :-)
allocation_alloc_other_private_memory : Lemma
  Forall(at : Allocation_Table(pm),
                        ac1, ac2 : (at`hierarchy`nodes), size : posnat) :
    pm`states(s) And
    allocation_ok?(pm)(at, s) And
    OK?(ac1`alloc(pm)(size)(s)) And
    Not data(ac1`alloc(pm)(size)(s)) = null_address And
    Not ac1 = ac2 Implies
      stays_unchanged(pm)(s, state(ac1`alloc(pm)(size)(s)),
                                                ac2`private_mem(pm))

% proof status :-)
% allocation_alloc_TCC1 : TCC Obligation

% proof status :-)
allocation_alloc : Lemma
```

```
    Forall(at : Allocation_Table(pm),
                          ac : (at`hierarchy`nodes), size : posnat) :
      pm`states(s) And
      allocation_ok?(pm)(at, s) And
      OK?(ac`alloc(pm)(size)(s)) And
      Not data(ac`alloc(pm)(size)(s)) = null_address Implies
        allocation_ok?(pm)(
                at WITH [`info(ac) :=
                    cons((data(ac`alloc(pm)(size)(s)), size), at`info(ac))],
                state(ac`alloc(pm)(size)(s)))

End Allocation_Table
```

B.3 bits.pvs

```
Bits : Theory
Begin
  Importing Alignment

  % MV: do we need the size limitats any more

  n, i : Var nat

  bool_to_nat(b : bool) : nat = if b then 1 else 0 endif

  % proof status :-)
  bool_to_nat_below : Judgement bool_to_nat(b : bool) HAS_TYPE below(2)

  nat_to_bool(n : below(2)) : bool = n = 1

  % proof status :-)
  nat_bool_iso : Lemma Forall(b : bool) : nat_to_bool(bool_to_nat(b)) = b

  % return true if the i-th bit in n is 1
  cut_bit(n, i) : bool = rem(2)(ndiv(n, expt(2, i))) = 1

  % proof status :-)
  cut_bit_zero : Lemma cut_bit(0, i) = False

  % cut k bits out of n, starting at i
  cut_bits(n, i, k : nat) : nat =
    rem(expt(2, k))(ndiv(n, expt(2, i)))

  % proof status :-)
  cut_bits_below : Judgement cut_bits(n, i, k : nat)
                          HAS_TYPE below(expt(2, k))

  % proof status :-)
  cut_bits_aligned : Lemma Forall(n, i, k : nat) :
    i <= k And cut_bits(n, 0, k) = 0 Implies aligned?(i)(n)

  % proof status :-)
```

```
aligned_cut_bits : Lemma Forall(n, i, j : nat, k : nat) :
  i + j <= k And aligned?(k)(n) Implies
    cut_bits(n, i, j) = 0

% proof status :-)
aligned_add_cut_bits : Lemma Forall(n, m, i, j, k : nat) :
  i + j <= k And aligned?(k)(n) Implies
    cut_bits(n + m, i, j) = cut_bits(m, i, j)

% proof status :-)
cut_bit_bits : Lemma
  cut_bit(n, i) = IF cut_bits(n, i, 1) = 1 Then true Else false Endif

% proof status :-)
cut_bits_zero : Lemma Forall(i, k : nat) : cut_bits(0, i, k) = 0

% proof status :-)
zero_cut_bits : Lemma Forall(n, i, k : nat) : k = 0 Implies
  cut_bits(n, i, k) = 0

% proof status :-)
% cut_bits_cut_bits_TCC1 : TCC Obligation

% proof status :-)
cut_bits_cut_bits : Lemma
  Forall(n, i1, k1, i2, k2 : nat) :
    cut_bits(cut_bits(n, i1, k1), i2, k2) =
      IF i2 >= k1 Then 0
      Elsif i2 + k2 > k1 Then cut_bits(n, i1 + i2, k1 - i2)
      Else cut_bits(n, i1 + i2, k2)
      Endif

% proof status :-)
cut_bit_cut_bits : Lemma
  Forall(n, i1, k1, i2 : nat) :
    cut_bit(cut_bits(n, i1, k1), i2) =
      IF i2 >= k1 Then False
      Else cut_bit(n, i1 + i2)
      Endif


shift_bits_left(n, i) : nat = n * expt(2, i)

% proof status :-)
% shift_bits_left_less_TCC1 : TCC Obligation

% proof status :-)
shift_bits_left_less : Lemma Forall(n, i, k : nat) :
  k >= i And n < expt(2, k - i) Implies
    shift_bits_left(n, i) < expt(2, k)

% proof status :-)
```

```
aligned_shift_bits_ok : Lemma Forall(n, i, j : nat) :
  i <= j Implies
    aligned?(i)(shift_bits_left(n, j))

% proof status :-)
% aligned_shift_bits_reduce_TCC1 : TCC Obligation

% proof status :-)
aligned_shift_bits_reduce : Lemma Forall(n, i, j : nat) :
  i > j And aligned?(i - j)(n) Implies
    aligned?(i)(shift_bits_left(n, j))

shift_bits_right(n, i) : nat = ndiv(n, expt(2, i))

% proof status :-)
% overwrite_bits_TCC1 : TCC Obligation

% overwrite_bits(n, m, i, j)
% replaces k bits starting from i in word n
% with k bits starting at i in word m
overwrite_bits(n, m, i, k : nat) : nat =
  n - rem(expt(2, i + k))(n)
      + shift_bits_left(cut_bits(m, i, k), i)
      + rem(expt(2, i))(n)

% proof status :-)
overwrite_bits_less : Lemma Forall(n, m, i, k, j : nat) :
  n < expt(2, j) And i + k <= j Implies
    overwrite_bits(n, m, i, k) < expt(2, j)

overwrite_bit(n, m, i : nat) : nat = overwrite_bits(n, m, i, 1)

% proof status :-)
overwrite_bit_less : Lemma Forall(n, m, i, k : nat) :
  n < expt(2, k) And i < k Implies
    overwrite_bit(n, m, i) < expt(2, k)


% overwrites bit i in n with b
overwrite_bool_bit(n : nat, b : bool, i : nat) : nat =
  overwrite_bit(n, shift_bits_left(bool_to_nat(b), i), i)

% proof status :-)
overwrite_bool_bit_less : Lemma Forall(n : nat, b : bool, i, k : nat) :
  n < expt(2, k) And i < k Implies
    overwrite_bool_bit(n, b, i) < expt(2, k)


% overwrite k bits in n at index i with the first k bits from m
overwrite_shift_bits(n, m, i : nat, k : nat) : nat =
  overwrite_bits(n, shift_bits_left(m, i), i, k)
```

```
% proof status :-)
overwrite_shift_bits_less : Lemma Forall(n, m, i, k, j : nat) :
  n < expt(2, j) And i + k <= j Implies
    overwrite_shift_bits(n, m, i, k) < expt(2, j)

% proof status :-)
aligned_overwrite_shift_bits_more : Lemma
  Forall(a, n, m, i : nat, k : posnat) :
    a <= i And aligned?(a)(n) Implies
      aligned?(a)(overwrite_shift_bits(n, m, i, k))


% proof status :-)
% aligned_overwrite_shift_bits_less_TCC1 : TCC Obligation

% the aligned(a)(n) is a very strong assumptions,
% closer to necessity would be
% aligned(a)(delete bits i to i + k -1 in n)
% proof status :-)
aligned_overwrite_shift_bits_less : Lemma
  Forall(a, n, m, i : nat, k : posnat) :
    i < a And aligned?(a)(n) And aligned?(a - i)(m) Implies
      aligned?(a)(overwrite_shift_bits(n, m, i, k))


% proof status :-)
% cut_bits_shift_bits_left_TCC1 : TCC Obligation

% proof status :-)
% cut_bits_shift_bits_left_TCC2 : TCC Obligation

% Don't install cut_bits_shift_bits_left as auto rewrite,
% it can be circular for ci = 0. Use the lemmas below instead.
% proof status :-)
cut_bits_shift_bits_left : Lemma Forall(n, si, ci, ck : nat) :
  cut_bits(shift_bits_left(n, si), ci, ck) =
    If ci + ck <= si Then 0
    Elsif ci < si
    Then cut_bits(shift_bits_left(n, si - ci), 0, ck)
    Else cut_bits(n, ci - si, ck)
    Endif

% proof status :-)
cut_bits_shift_bits_left_outside : Lemma
  Forall(n, si, ci, ck : nat) :
    ci + ck <= si Implies
      cut_bits(shift_bits_left(n, si), ci, ck) = 0

% proof status :-)
% cut_bits_shift_bits_left_overlap_TCC1 : TCC Obligation
```

```
% proof status :-)
cut_bits_shift_bits_left_overlap : Lemma
  Forall(n, si, ci, ck : nat) :
    ci + ck > si And ci < si And ci > 0 Implies
      cut_bits(shift_bits_left(n, si), ci, ck) =
        cut_bits(shift_bits_left(n, si - ci), 0, ck)

% proof status :-)
cut_bits_shift_bits_left_in : Lemma
  Forall(n, si, ci, ck : nat) :
    ci >= si Implies
      cut_bits(shift_bits_left(n, si), ci, ck) =
        cut_bits(n, ci - si, ck)


% proof status :-)
overwrite_shift_bits_zero : Lemma
  Forall(z, m, i, k : nat) :
    z = 0 Implies
      overwrite_shift_bits(z, m, i, k) =
        shift_bits_left(cut_bits(m, 0, k), i)

% proof status :-)
zero_overwrite_shift_bits : Lemma
  Forall(n, m, i, k : nat) : k = 0 Implies
      overwrite_shift_bits(n, m, i, k) = n

% proof status :-)
% bit_split_TCC1 : TCC Obligation

% proof status :-)
bit_split : Lemma Forall(n, i, k : nat) :
  k <= i And n < expt(2, i) Implies
    n = overwrite_shift_bits(cut_bits(n, 0, k),
                             cut_bits(n, k, i - k), k, i - k)

% proof status :-)
shift_bits_left_cut_bits : Lemma Forall(n, i, k, j : nat) :
  i = j And aligned?(i)(n) And n < expt(2, i + k) Implies
    shift_bits_left(cut_bits(n, i, k), j) = n

% proof status :-)
cut_bits_0_to_below : Lemma Forall(n, k : nat) :
  n < expt(2, k) Implies cut_bits(n, 0, k) = n

% proof status :-)
cut_bits_overwrite_bits_disjoined : Lemma
  Forall(n, m, oi, ok : nat, ci, ck : nat) :
    oi + ok <= ci Or ci + ck <= oi Implies
      cut_bits(overwrite_bits(n, m, oi, ok), ci, ck) = cut_bits(n, ci, ck)

% proof status :-)
```

```
cut_bits_overwrite_bool_bit_disjoined : Lemma
  Forall(n : nat, b : bool, oi, ci, ck : nat) :
    oi < ci Or ci + ck <= oi Implies
      cut_bits(overwrite_bool_bit(n, b, oi), ci, ck) = cut_bits(n, ci, ck)


% proof status :-)
cut_bits_overwrite_bits_contained : Lemma
  Forall(n, m, oi, ok, ci, ck : nat) :
    oi <= ci And ci + ck <= oi + ok Implies
      cut_bits(overwrite_bits(n, m, oi, ok), ci, ck) =
        cut_bits(m, ci, ck)

% proof status :-)
cut_bits_overwrite_shift_bits_disjoint : Lemma
  Forall(n, m, i1, k1, i2, k2 : nat) :
    i1 + k1 <= i2 OR i2 + k2 <= i1 Implies
      cut_bits(overwrite_shift_bits(n, m, i1, k1), i2, k2) =
              cut_bits(n, i2, k2)

% proof status :-)
% cut_bits_overwrite_shift_bits_contained_TCC1 : TCC Obligation

% proof status :-)
cut_bits_overwrite_shift_bits_contained : Lemma
  Forall(n, m, i1, k1, i2, k2 : nat) :
    i1 <= i2 And i2 + k2 <= i1 + k1 Implies
      cut_bits(overwrite_shift_bits(n, m, i1, k1), i2, k2) =
              cut_bits(m, i2 - i1, k2)

% proof status :-)
% cut_bit_overwrite_shift_bits_TCC1 : TCC Obligation

% proof status :-)
cut_bit_overwrite_shift_bits : Lemma
  Forall(n, m, i1, k1, i2 : nat) :
    cut_bit(overwrite_shift_bits(n, m, i1, k1), i2) =
      IF i1 <= i2 And i2 < i1 + k1
      Then cut_bit(m, i2 - i1)
      Else cut_bit(n, i2)
      Endif

% not needed any more, proof saved below
% cut_bits_overwrite_bool_bit_contained : Lemma
% Forall(n : nat, b : bool, oi : nat, ci, ck : nat) :
% ck = 1 And oi = ci Implies
% cut_bits(overwrite_bool_bit(n, b, oi), ci, ck) =
% bool_to_nat(b)

% proof status :-)
cut_bit_overwrite_bool_bit : Lemma
  Forall(n : nat, b : bool, i, j : nat) :
```

```
      cut_bit(overwrite_bool_bit(n, b, i), j) =
        IF i = j Then b
        Else cut_bit(n, j)
        Endif

  % proof status :-)
  overwrite_shift_bits_merge : Lemma
    Forall(m, ci1, ck1, si, ci2, ck2, os, ok : nat) :
      ci1 + ck1 = ci2 And
      si + ck1 = os And
      ck2 = ok
        Implies
          overwrite_shift_bits(shift_bits_left(cut_bits(m, ci1, ck1), si),
                               cut_bits(m, ci2, ck2), os, ok) =
          shift_bits_left(cut_bits(m, ci1, ck1 + ck2), si)

End Bits


% Bit operations for below[2^p]
Bitops[p : posnat] : Theory
Begin

  Importing Bits, More_List_Props

  n : Var nat
  val : Var nat
  v1, v2 : Var nat

  bit_set?(n, val) : bool = cut_bit(val, n)

  % proof status :-)
  bit_set_zero_mask : Lemma
    Forall (n : nat) : Not bit_set?(n, 0)

  % proof status :-)
  bit_set_ndiv_expt2 : Lemma
    Forall (n, val, m : nat) :
      bit_set?(n, ndiv(val, expt(2, m))) = bit_set?(n + m, val)

  % proof status :-)
  % bit_set_full_mask_TCC1 : TCC Obligation

  % proof status :-)
  bit_set_full_mask : Lemma
    Forall (n : nat) : n < p Implies bit_set?(n, expt(2, p) - 1)

  % proof status :-)
  bits_equal : Lemma
    Forall (v1, v2, p : nat) :
      (Forall (n : nat) : n < p Implies bit_set?(n, v1) = bit_set?(n, v2)) Implies
        rem(expt(2, p))(v1) = rem(expt(2, p))(v2)
```

```
% proof status :-)
bits_equal2 : Lemma
  Forall (v1, v2 : below[expt(2, p)]) :
    (Forall (n : nat) : n < p Implies bit_set?(n, v1) = bit_set?(n, v2)) IFF
      v1 = v2

% proof status :-)
bits_equal3 : Lemma
  Forall (v1, v2 : nat) :
    (Forall (n : nat) : bit_set?(n, v1) = bit_set?(n, v2)) Implies
      v1 = v2

% proof status :-)
bits_equal4 : Lemma
  Forall (v1, v2 : nat, m : nat) :
    (Forall (n : nat) : n >= m Implies bit_set?(n, v1) = bit_set?(n, v2)) Implies
      ndiv(v1, expt(2, m)) = ndiv(v2, expt(2, m))

set_bit(n, val) : nat =
  If bit_set?(n, val) Then
    val
  Else
    val + expt(2,n)
  Endif

% proof status :-)
% clear_bit_TCC1 : TCC Obligation

clear_bit(n, val) : nat =
  If bit_set?(n, val) Then
    val - expt(2,n)
  Else
    val
  Endif

change_bit(val : nat, c : [nat -> bool])(n : nat) : nat =
  If c(n) Then
    set_bit(n, val)
  Else
    clear_bit(n, val)
  Endif

% proof status :-)
change_bit_single_bit_op : Lemma
  Forall (n, m, v : nat, c : [nat -> bool]) : n /= m Implies
    bit_set?(n, v) = bit_set?(n, change_bit(v, c)(m))

% proof status :-)
change_bit_result : Lemma
  Forall (n, v : nat, c : [nat -> bool]) :
    bit_set?(n, change_bit(v, c)(n)) = c(n)
```

```
% proof status :-)
% change_bits_TCC1 : TCC Obligation

% proof status :-)
% change_bits_TCC2 : TCC Obligation

change_bits(v : nat, c : [nat -> bool])(n : nat) : Recursive nat =
  If n = 0 Then
    change_bit(v, c)(0)
  Else
    change_bit(change_bits(v, c)(n - 1), c)(n)
  Endif
 Measure n

% proof status :-)
change_bits_result : Lemma
  Forall (m, n : nat, val : nat, c : [nat -> bool]) : m <= n Implies
    bit_set?(m, change_bits(val, c)(n)) = c(m)

% proof status :-)
change_bits_unchanged : Lemma
  Forall (m, n : nat, val : nat, c : [nat -> bool]) : m > n Implies
    bit_set?(m, change_bits(val, c)(n)) = bit_set?(m, val)

% proof status :-)
% change_bits_below_TCC1 : TCC Obligation

% proof status :-)
change_bits_below : Lemma
  Forall (v : below[expt(2, p)], c : [nat -> bool]) :
    change_bits(v, c)(p-1) < expt(2, p)

% proof status :-)
% binary_not_TCC1 : TCC Obligation

% binary ops
%------------
binary_not(val : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(val, Lambda (n : nat) : Not bit_set?(n, val))(p - 1)

% proof status :-)
binary_not_not : Lemma
  Forall ( val : below[expt(2, p)]) :
    binary_not(binary_not(val)) = val

% proof status :-)
% binary_and_TCC1 : TCC Obligation

binary_and(v1, v2 : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(v1, Lambda (n : nat) : bit_set?(n, v1) And bit_set?(n, v2))(p - 1)
```

```
% proof status :-)
binary_and_not : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_and(val, binary_not(val)) = 0

% proof status :-)
% binary_and_full_TCC1 : TCC Obligation

% proof status :-)
binary_and_full : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_and(val, expt(2, p) - 1) = val

% proof status :-)
% binary_and_zero_TCC1 : TCC Obligation

% proof status :-)
binary_and_zero : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_and(val, 0) = 0

% proof status :-)
% binary_or_TCC1 : TCC Obligation

binary_or(v1, v2 : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(v1, Lambda (n : nat) : bit_set?(n, v1) Or bit_set?(n, v2))(p - 1)

% proof status :-)
binary_or_not : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_or(val, binary_not(val)) = expt(2, p) - 1

% proof status :-)
% binary_xor_TCC1 : TCC Obligation

binary_xor(v1, v2 : below[expt(2, p)]) : below[expt(2, p)] =
  change_bits(v1, Lambda (n : nat) : bit_set?(n, v1) /= bit_set?(n, v2))(p - 1)

% proof status :-)
binary_xor_self : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_xor(val, val) = 0

% proof status :-)
binary_or_full : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_or(val, expt(2, p) - 1) = expt(2, p) - 1

% proof status :-)
binary_or_zero : Lemma
  Forall (val : below[expt(2, p)]) :
    binary_or(val, 0) = val
```

```
End Bitops

Bitlist_ops : Theory
Begin

  Importing General, Bitops[bits_per_byte]

  % Lifting to lists
  %==================

  % proof status :-)
  % binary_not_TCC1 : TCC Obligation

  binary_not(l : list[Byte]) : list[Byte] =
    map[Byte, Byte](binary_not)(l)

  % proof status :-)
  % binary_and_TCC1 : TCC Obligation

  binary_and(l1, l2 : list[Byte]) : list[Byte] =
    map2[Byte](binary_and)(l1, l2)

  % proof status :-)
  % binary_or_TCC1 : TCC Obligation

  binary_or(l1, l2 : list[Byte]) : list[Byte] =
    map2[Byte](binary_or)(l1, l2)

  % proof status :-)
  % binary_xor_TCC1 : TCC Obligation

  binary_xor(l1, l2 : list[Byte]) : list[Byte] =
    map2[Byte](binary_xor)(l1, l2)

  full_mask?(size : nat)(l : list[Byte]) : bool =
    length(l) = size And every(Lambda (t : Byte) : t = max_byte - 1)(l)

  zero_mask?(size : nat)(l : list[Byte]) : bool =
    length(l) = size And every(Lambda (t : Byte) : t = 0)(l)

  % proof status :-)
  singleton_zero_mask : Lemma
    Forall (s : nat) :
      singleton?(zero_mask?(s))

  % proof status :-)
  % bit_set?_TCC1 : TCC Obligation

  % proof status :-)
  % bit_set?_TCC2 : TCC Obligation
```

```
  bit_set?(n : nat, l : list[Byte]) : bool =
    If n < length(l) * max_byte Then
      bit_set?(rem(max_byte)(n), nth(l, ndiv(n, max_byte)))
    Else
      false
    Endif

  apply_bitop_valuefield(mask : list[Byte], bitop : [list[Byte] -> list[Byte]])(l : list[Byte]) : list[Byte] =
    binary_or(binary_and(l, binary_not(mask)), binary_and(bitop(l), mask))

End Bitlist_ops
```

B.4 challenge-linear.pvs

```
% MV: there are a few assumptions in the address in page table range stuff
%
% Current Address Space Layout (hypervisor/include/memory.h)
%
% | Global Area | CPU Local Area | AS Loc|
% |-------------------------------------------------------------------------
% |------- ------------------- -- --- ------ ----- ----- -- ------ -------
% | Kcode | Mem Pool | |RM|VGA|IOAPIC| Loc| |LAPIC|SP|CPULoc| IOPBM |
% |------- ------------------- -- --- ------ ----- ----- -- ------ -------
% |-------------------------------------------------------------------------
%
% HV: Assumptions
% - The global area is synced at the master PDIR level (fault.cpp/paging.cpp).
% - Master PDIR is in Kcode (.bss segment)
%
% Model:
% - address_in_page_table_range gives all page table entries used for the given
% address range. Page table entries outside this range may be in plain memory
% - this should at least allow to enter the fault handler without loosing plain
% memory because subset(ro_PF u rw_PF, ro u rw)
% ? is this a generic result for plain memory, i.e., pm(range) => pm(sub_range)
% and if so do we have it.

Linear_Memory_Blessing[% the underlying physical memory
                       Physical_memory : Type,
                       (Importing Plain_Memory[Physical_memory])
                       pm_phy : Plain_Memory] : Theory
Begin

  Importing Linear_Memory[Physical_memory, pm_phy],
            Memory_Change, Control_Register_Datatype

  oact : Var PRED[[Linear_memory -> SuperResult[Linear_memory]]]
  ro_addr : Var PRED[Memory_Address_4G]
  rw_addr : Var PRED[Memory_Address_4G]
  states : Var PRED[Linear_memory]

  linear_range : Var PRED[Memory_Address_4G]
```

```
s : Var Linear_memory

lvl : Var Level


% Sharing
%=========

% tuples of linear addresses that share the same physical address
linear_shared?(s)(a1, a2 : Memory_Address_4G) : bool =
  Exists(ac1, ac2 : Memory_access) :
    OK?(linear_resolve(a1, ac1)(s)) And
      OK?(linear_resolve(a2, ac2)(s)) And
        data(linear_resolve(a1, ac1)(s)) =
          data(linear_resolve(a2, ac2)(s))

% Address In Page Table
%======================

% proof status :-)
% PTab_Address_TCC1 : TCC Obligation

PTab_Address : Type =
  {a : Memory_Address_4G | aligned?(bits_per_level + pe_size)(offset(a))}

% proof status :-)
% pe_in_pdir_range?_TCC1 : TCC Obligation

pe_in_pdir_range?(s, linear_range)(phy_a : Memory_Address_4G) : bool =
  Let res = read_data(pm_phy, pdbr_data_type)(PDBR)(s) In
    OK?(res) And
      Exists (lin_a : (linear_range)) :
        xlat_idx(pdir_lvl, data(res)`base_addr, lin_a) = phy_a

% proof status :-)
% pe_in_ptab_range?_TCC1 : TCC Obligation

% proof status :-)
% pe_in_ptab_range?_TCC2 : TCC Obligation

% proof status :-)
% pe_in_ptab_range?_TCC3 : TCC Obligation

% proof status :-)
% pe_in_ptab_range?_TCC4 : TCC Obligation

pe_in_ptab_range?(s, linear_range)(phy_a : Memory_Address_4G) : bool =
  Exists (pde : (pe_in_pdir_range?(s, linear_range))) :
    Exists (lin_a : (linear_range)) :
      Let res = read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s) In
        OK?(res) And pde_pt?(pdir(data(res))) And
```

```
            xlat_idx(ptab_lvl, base(data(res)), lin_a) = phy_a

  % proof status :-)
  % pe_in_pt_range?_TCC1 : TCC Obligation

  % proof status :-)
  % pe_in_pt_range?_TCC2 : TCC Obligation

  pe_in_pt_range?(s, linear_range, lvl)(phy_a : Memory_Address_4G) : bool =
    Cond
      lvl = pdir_lvl -> pe_in_pdir_range?(s, linear_range)(phy_a),
      lvl = ptab_lvl -> pe_in_ptab_range?(s, linear_range)(phy_a)
    EndCond

  % proof status :-)
  pe_in_pt_range_aligned : Lemma
    Forall (phy_a : Memory_Address_4G) :
      pe_in_pt_range?(s, linear_range, lvl)(phy_a) Implies
        aligned?(pe_size)(offset(phy_a))

  address_in_pt_range?(s, linear_range)(phy_a : Memory_Address_4G) : bool =
    Exists (pe : Memory_Address_4G, lvl : Level) :
      pe_in_pt_range?(s, linear_range, lvl)(pe) And
        address_block(pe, expt(2, pe_size))(phy_a)

  % proof status :-)
  pe_in_pt_address_in_pt : Lemma
    Forall (base : Memory_Address_4G) :
      pe_in_pt_range?(s, linear_range, lvl)(base)
    Implies
      subset?(address_block(base, expt(2, pe_size)),
              address_in_pt_range?(s, linear_range))

  % Physical Addresses
  %====================

  virt_to_phys_range(s : Linear_memory, linear_range : PRED[Memory_Address_4G],
                     mem_ac : PRED[Memory_access])
                    (phy_a : Memory_Address_4G) : bool =
    Exists (lin_a : (linear_range), ac : (mem_ac)) :
      Let res = linear_resolve(lin_a, ac)(s) In
        OK?(res) And data(res) = phy_a

  virt_to_phys_range(s, linear_range) : [Memory_Address_4G -> bool] =
    virt_to_phys_range(s, linear_range, fullset[Memory_access])

  % Linear Resolve Transformers
  %============================

  linear_resolve_register_transformers
              (q : [Physical_memory -> SuperResult[Physical_memory]]) : bool =
    q = expr_2_super(read_data(pm_phy, segment_reg_data_type)(CS)) Or
```

```
  q = expr_2_super(read_data(pm_phy, pdbr_data_type)(PDBR)) Or
  Exists (a : Memory_Address_4G) :
    q = expr_2_super(write_data(pm_phy, address_data_type)(CR2, a))

% Linear Blessing
%=================

% Those page table entries that are used to translate the address
% range for which linear memory is defined are not part of the
% linear blessed range. This is because as part of the translation,
% reference bits in this area may be updated. Other PTEs not involved
% in the translation may well be part of plain memory.
%
% This lets the page fault handler work inside plain memory and update
% the active kernel page table in all areas besides the one in which the
% page-fault handler itself runs.

% make in memory explicit requirement !!!
linear_blessed?(s, ro_addr, rw_addr) : bool =
  % resolve OK?
  (Forall (a : Memory_Address_4G) : union(ro_addr, rw_addr)(a) Implies
    OK?(linear_resolve(a, Read)(s)) And OK?(linear_resolve(a, Execute)(s)))
  And
  (Forall (a : Memory_Address_4G) : rw_addr(a) Implies
    OK?(linear_resolve(a, Write)(s))) And
  % underlying blessing memory read addresses
  subset?(virt_to_phys_range(s, union(ro_addr, rw_addr),
                                union(singleton(Read), singleton(Execute))),
                                union(pm_phy`ro_addr, pm_phy`rw_addr))
  And
  subset?(virt_to_phys_range(s, rw_addr, singleton(Write)), pm_phy`rw_addr)
  And
  subset?(address_in_pt_range?(s, union(ro_addr, rw_addr)), pm_phy`rw_addr)
  And
  % disjoint mapping of (ro_addr, rw_addr) and ptabs
  disjoint?(virt_to_phys_range(s, union(ro_addr, rw_addr)),
            address_in_pt_range?(s, union(ro_addr, rw_addr)))
  And
  % not shared
  % blessing memory read, memory_write addresses if these are disjoint
  (Forall (a1, a2 : Memory_Address_4G) :
     rw_addr(a1) And union(ro_addr, rw_addr)(a2) Implies
       Not linear_shared?(s)(a1, a2)) And
  % linear memory side effects
  True % no side effect restrictions

% proof status :-)
% is_linear_plain_memory?_TCC1 : TCC Obligation

% proof status :-)
% is_linear_plain_memory?_TCC2 : TCC Obligation
```

```
  % proof status :-)
  % is_linear_plain_memory?_TCC3 : TCC Obligation

  % proof status :-)
  % is_linear_plain_memory?_TCC4 : TCC Obligation

  % proof status :-)
  % is_linear_plain_memory?_TCC5 : TCC Obligation

  % proof status :-)
  % is_linear_plain_memory?_TCC6 : TCC Obligation

  is_linear_plain_memory?(pm : Plain_Memory[Linear_memory]) : bool =
    % linear pm
    pm`mem = linear_pm And
    % unmodified page-table area
      % cs
      (Forall (s : (pm`states)) :
         OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))) And
      (Forall (s1, s2 : (pm`states)) :
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
           data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))) And
      % pdbr
      (Forall (s : (pm`states)) :
         OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))) And
      (Forall (s1, s2 : (pm`states)) :
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
           data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))) And
      % pdir range
      (Forall (s1, s2 : (pm`states)) :
         pe_in_pdir_range?(s1, union(pm`ro_addr, pm`rw_addr)) =
           pe_in_pdir_range?(s2, union(pm`ro_addr, pm`rw_addr)) And
         pe_in_ptab_range?(s1, union(pm`ro_addr, pm`rw_addr)) =
           pe_in_ptab_range?(s2, union(pm`ro_addr, pm`rw_addr))) And
      % page tables
      (Forall (s : (pm`states), lvl : Level,
                 a : (pe_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr), lvl))) :
         OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))) And
      (Forall (s1, s2 : (pm`states), lvl : Level,
                 a : (pe_in_pt_range?(s1, union(pm`ro_addr, pm`rw_addr), lvl))) :
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)),
                      Write) =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)),
                      Write)) And
    % underlying blessed memory
    plain_memory?(pm_phy) And
    % underlying states
    pm`states = pm_phy`states And
    % underlying other actions
    subset?(pm`other_actions, pm_phy`other_actions) And
    subset?(linear_resolve_register_transformers, pm_phy`other_actions) And
    % memory address range
```

```
    (Forall (a : (union(pm`ro_addr, pm`rw_addr))) :
       Mem?(type_of(a)) And 0 <= offset(a) And
       offset(a) < max_linear_offset) And
    % linear_blessed states
    (Forall (s : (pm`states)) : linear_blessed?(s, pm`ro_addr, pm`rw_addr))

  %%%%%%%%% Plain Memory Structure for Linear Memory %%%%%%%%%%%

  linear_plain_memory(states, oact, ro_addr, rw_addr)
                                    : Plain_Memory[Linear_memory] =
  (#
    mem := linear_pm,
    states := states,
    ro_addr := ro_addr,
    rw_addr := rw_addr,
    other_actions := oact
  #)

End Linear_Memory_Blessing

%>> MV:
% - todo: establish is_linear_plain_memory? from a single state and
% conclude that applying the state transformers yields new
% states for which is_linear_plain_memory? holds.

Linear_Memory_Properties[Physical_memory : Type,
                         (Importing Plain_Memory[Physical_memory])
                         pm_phy : Plain_Memory] : Theory
Begin

  Importing Linear_Memory_Blessing[Physical_memory, pm_phy],
            Plain_Mem_Rewrites,
            Even_More_List_Props

  s : Var Linear_memory
  lvl : Var Level
  pm : Var Plain_Memory[Linear_memory]

  % Helpers
  %=========

  % rewrite memory_read(pm`mem...) = linear_read(...)
  % proof status :-)
  pm_read_linear : Lemma
    Forall (a : Address) :
      is_linear_plain_memory?(pm) Implies
        memory_read(pm`mem)(a) = linear_read(a)

  % proof status :-)
  pm_write_linear : Lemma
    Forall (a : Address, b : Byte) :
      is_linear_plain_memory?(pm) Implies
```

```
        memory_write(pm`mem)(a, b) = linear_write(a, b)

  % proof status :-)
  pm_read_side_effect_linear : Lemma
    Forall (a : Address, bl : list[Byte], cp : bool) :
      is_linear_plain_memory?(pm) Implies
        memory_read_side_effect(pm`mem)(a, bl, cp) =
          linear_read_side_effect(a, bl, cp)

  % proof status :-)
  pm_write_side_effect_linear : Lemma
    Forall (a : Address, bl : list[Byte], cp : bool) :
      is_linear_plain_memory?(pm) Implies
        memory_write_side_effect(pm`mem)(a, bl, cp) =
          linear_write_side_effect(a, bl, cp)

  % proof status :-)
  pm_states : Lemma
    is_linear_plain_memory?(pm) Implies pm`states = pm_phy`states

  % proof status :-)
  pm_plain_phy : Lemma
    is_linear_plain_memory?(pm) Implies plain_memory?(pm_phy)

  % proof status :-)
  % pm_memory_addr_TCC1 : TCC Obligation

  % proof status :-)
  % pm_memory_addr_TCC2 : TCC Obligation

  % proof status :-)
  pm_memory_addr : Lemma
    Forall (a : Address) :
      is_linear_plain_memory?(pm) And union(pm`ro_addr, pm`rw_addr)(a) Implies
        in_memory(min_linear, max_linear)(a) And Mem?(type_of(a))

  % proof status :-)
  pm_linear_blessed : Lemma
    is_linear_plain_memory?(pm) And pm`states(s) Implies
      linear_blessed?(s, pm`ro_addr, pm`rw_addr)

  % proof status :-)
  % pm_linear_resolve_read_ok_TCC1 : TCC Obligation

  % proof status :-)
  pm_linear_resolve_read_ok : Lemma
    Forall (s : (pm`states), a : (union(pm`ro_addr, pm`rw_addr))) :
      is_linear_plain_memory?(pm) Implies OK?(linear_resolve(a, Read)(s))

  % proof status :-)
  % pm_linear_resolve_write_ok_TCC1 : TCC Obligation
```

```
% proof status :-)
pm_linear_resolve_write_ok : Lemma
  Forall (s : (pm`states), a : (pm`rw_addr)) :
    is_linear_plain_memory?(pm) Implies OK?(linear_resolve(a, Write)(s))

% proof status :-)
pm_linear_resolve_reg_transformers_other : Lemma
  is_linear_plain_memory?(pm)
    Implies
      subset?(linear_resolve_register_transformers, pm_phy`other_actions)

% proof status :-)
pm_unchanged_singleton_linear_resolve_reg_transformers : Lemma
  Forall (q : [Physical_memory -> SuperResult[Physical_memory]]) :
    is_linear_plain_memory?(pm) And linear_resolve_register_transformers(q)
      Implies
    unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(q),
                                union(pm_phy`ro_addr, pm_phy`rw_addr))

% proof status :-)
pm_address_in_pt_range_in_rw_addr : Lemma
  is_linear_plain_memory?(pm) And pm`states(s) Implies
    subset?(address_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr)),
            pm_phy`rw_addr)

% proof status :-)
% address_block_in_memory_TCC1 : TCC Obligation

% proof status :-)
address_block_in_memory : Lemma
  Forall (a : Address, bl : list[Byte]) :
  is_linear_plain_memory?(pm) And
    subset?(address_block(a, length(bl)), union(pm`ro_addr, pm`rw_addr)) And
      cons?(bl)
   Implies
     a + length(bl) <= reg_size(max_linear)(type_of(a))

% proof status :-)
address_block_in_memory2 : Lemma
  Forall (a : Address, bl : list[Byte]) :
  is_linear_plain_memory?(pm) And
    subset?(address_block(a, length(bl)), union(pm`ro_addr, pm`rw_addr)) And
      cons?(bl)
   Implies
     0 <= a`offset

% proof status :-)
raise_fault_transformer_invariant : Lemma
  Forall (priv : Memory_privilege, access : Memory_access, present : bool,
          pfa : Memory_Address_4G) :
    is_linear_plain_memory?(pm) Implies
      transformer_invariant?(pm_phy`states, singleton(expr_2_super(
```

```
                                raise_fault(priv, access, present, pfa))))

  % proof status :-)
  raise_fault_unchanged_memory_invariant : Lemma
    Forall (priv : Memory_privilege, access : Memory_access, present : bool,
              pfa : Memory_Address_4G) :
      is_linear_plain_memory?(pm) Implies
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
          singleton(expr_2_super(raise_fault(priv, access, present, pfa))),
            union(pm_phy`ro_addr, pm_phy`rw_addr))

  % Side Effects
  %==============
  % proof status :-)
  apply_side_effects_pm_states : Lemma
    Forall (l : list[[Memory_Address_4G, list[Byte]]],
              f : [Memory_Address_4G, list[Byte] ->
                     [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]]) :
      is_linear_plain_memory?(pm) And
        every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
          transformer_invariant?(pm`states, singleton(expr_2_super(f(e)))))(l)
      Implies
        transformer_invariant?(pm`states, singleton(
                                  expr_2_super(apply_side_effects(l, f))))

  % proof status :-)
  apply_side_effects_ok : Lemma
    Forall (l : list[[Memory_Address_4G, list[Byte]]],
              f : [Memory_Address_4G, list[Byte] ->
                     [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]]) :
      is_linear_plain_memory?(pm) And
        pm`states(s) And
          (Forall (s1 : (pm`states)) :
             every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
               OK?(f(e)(s1)))(l)) And
             every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
               transformer_invariant?(pm`states,
                                        singleton(expr_2_super(f(e)))))(l)
      Implies
        OK?(apply_side_effects(l, f)(s))

  % proof status :-)
  % apply_side_effects_same_result_TCC1 : TCC Obligation

  % proof status :-)
  apply_side_effects_same_result : Lemma
    Forall (l : list[[Memory_Address_4G, list[Byte]]],
              f : [Memory_Address_4G, list[Byte] ->
                     [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]]) :
      is_linear_plain_memory?(pm) And
        pm`states(s) And
        (Forall (s1 : (pm`states)) :
```

```
          every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
                OK?(f(e)(s1)))(l)) And
      (Forall (s1 : (pm`states)) :
         every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
                OK?(f(e)(s1)) Implies data(f(e)(s1)) = e`2)(l)) And
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
        transformer_invariant?(pm`states, singleton(expr_2_super(f(e)))))(l)
    Implies
      data(apply_side_effects(l, f)(s)) =
        reduce(null, Lambda (e : [Address, list[Byte]], tail : list[Byte]) :
          Let head = e`2 In append(head, tail))(l)

% proof status :-)
apply_side_effects_unchanged : Lemma
  Forall (l : list[[Memory_Address_4G, list[Byte]]],
          f : [Memory_Address_4G, list[Byte] ->
                 [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]],
          addresses : PRED[Memory_Address_4G]) :
    is_linear_plain_memory?(pm) And
    subset?(addresses, union(pm_phy`ro_addr, pm_phy`rw_addr)) And
    every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
      unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                singleton(expr_2_super(f(e))), addresses))(l)
    Implies
      unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                singleton(expr_2_super(apply_side_effects(l, f))), addresses)

% Main Resolve Results
%=====================

% proof status :-)
subset_ptab_address_rw_addr : Lemma
  Forall (base : Memory_Address_4G) :
    subset?(address_in_pt_range?(s,
              union(restrict[Address, Memory_Address_4G, bool](pm`ro_addr),
                    restrict[Address, Memory_Address_4G, bool](pm`rw_addr))),
              restrict[Address, Memory_Address_4G, bool](pm_phy`rw_addr)) And
    subset?(address_block(base, expt(2, pe_size)),
              extend[Address, Memory_Address_4G, bool, FALSE]
              (address_in_pt_range?(s,
                                    restrict[Address, Memory_Address_4G, bool]
                                    (union(pm`ro_addr, pm`rw_addr)))))
  Implies
    subset?(address_block(base, expt(2, pe_size)), pm_phy`rw_addr)


% proof status :-)
translate_transformer_invariant : Lemma
  Forall (base : PTab_Address, lin_a : Memory_Address_4G,
          access : Memory_access, priv : Memory_privilege) :
  union(pm`ro_addr, pm`rw_addr)(lin_a) And
    is_linear_plain_memory?(pm) And
```

```
        subset?(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)),
                pm_phy`rw_addr)
    Implies
      transformer_invariant?(pm`states, singleton(expr_2_super(
                                translate(lvl, base, lin_a, access, priv))))

  % proof status :-)
  % translate_unchanged_pm_phy_except_pe_TCC1 : TCC Obligation

  % proof status :-)
  % translate_unchanged_pm_phy_except_pe_TCC2 : TCC Obligation

  % proof status :-)
  translate_unchanged_pm_phy_except_pe : Lemma
    Forall (base : PTab_Address, lin_a : Memory_Address_4G,
            access : Memory_access, priv : Memory_privilege) :
    union(pm`ro_addr, pm`rw_addr)(lin_a) And
      is_linear_plain_memory?(pm) And
        subset?(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)),
                pm_phy`rw_addr) And
      (Forall (s : (pm`states)) :
         OK?(read_data(pm_phy, paging_data_type(lvl))
                       (xlat_idx(lvl, base, lin_a))(s))) And
      (Forall (s1, s2 : (pm`states)) :
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))
                           (xlat_idx(lvl, base, lin_a))(s1)), Write) =
          set_reference(data(read_data(pm_phy, paging_data_type(lvl))
                             (xlat_idx(lvl, base, lin_a))(s2)), Write))
    Implies
      Let pe_addr = xlat_idx(lvl, base, lin_a) In
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
          singleton(expr_2_super(translate(lvl, base, lin_a, access, priv))),
          difference(union(pm_phy`ro_addr, pm_phy`rw_addr),
                     address_block(pe_addr, expt(2, pe_size))))

  % proof status :-)
  % translate_result_ok_TCC1 : TCC Obligation

  % proof status :-)
  translate_result_ok : Lemma
    Forall (s1, s2 : (pm`states), base : PTab_Address,
            lin_a : Memory_Address_4G, access : Memory_access,
            priv : Memory_privilege) :
    union(pm`ro_addr, pm`rw_addr)(lin_a) And
      is_linear_plain_memory?(pm) And
        pe_in_pt_range?(s1, union(pm`ro_addr, pm`rw_addr), lvl)
                        (xlat_idx(lvl, base, lin_a)) And
          OK?(translate(lvl, base, lin_a, access, priv)(s1))
    Implies
      OK?(translate(lvl, base, lin_a, access, priv)(s2))

  % proof status :-)
```

```
% translate_result_as_read_TCC1 : TCC Obligation

% proof status :-)
% translate_result_as_read_TCC2 : TCC Obligation

% proof status :-)
translate_result_as_read : Lemma
  Forall (base : PTab_Address, lin_a : Memory_Address_4G,
          access : Memory_access, priv : Memory_privilege) :
  union(pm`ro_addr, pm`rw_addr)(lin_a) And
    is_linear_plain_memory?(pm) And
      OK?(translate(lvl, base, lin_a, access, priv)(s))
  Implies
    Let res = read_data(pm_phy, paging_data_type(lvl))
                       (xlat_idx(lvl, base, lin_a))(s) In
      data(translate(lvl, base, lin_a, access, priv)(s)) =
        (is_leaf?(data(res)), base(data(res)))

% proof status :-)
translate_same_result : Lemma
  Forall (s1, s2 : (pm`states), base : PTab_Address,
          lin_a : Memory_Address_4G, ac1, ac2 : Memory_access,
          priv : Memory_privilege) :
    union(pm`ro_addr, pm`rw_addr)(lin_a) And
      is_linear_plain_memory?(pm) And
        pe_in_pt_range?(s1, union(pm`ro_addr, pm`rw_addr), lvl)
                       (xlat_idx(lvl, base, lin_a)) And
          OK?(translate(lvl, base, lin_a, ac1, priv)(s1)) And
            OK?(translate(lvl, base, lin_a, ac2, priv)(s2))
  Implies
    data(translate(lvl, base, lin_a, ac1, priv)(s1)) =
      data(translate(lvl, base, lin_a, ac2, priv)(s2))

% proof status :-)
% xlat_pe_in_pdir_range_TCC1 : TCC Obligation

% proof status :-)
% xlat_pe_in_pdir_range_TCC2 : TCC Obligation

% proof status :-)
% xlat_pe_in_pdir_range_TCC3 : TCC Obligation

% proof status :-)
xlat_pe_in_pdir_range : Lemma
  Forall (addr : Memory_Address_4G, s1, s2 : Linear_memory) :
    union(pm`ro_addr, pm`rw_addr)(addr) And
      is_linear_plain_memory?(pm) And
        pm`states(s1) And pm`states(s2)
  Implies
    pe_in_pdir_range?(s1, union(pm`ro_addr, pm`rw_addr))
      (xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)
                                        (PDBR)(s2))`base_addr, addr))
```

```
  % proof status :-)
  % xlat_pe_in_pt_range_TCC1 : TCC Obligation

  % proof status :-)
  % xlat_pe_in_pt_range_TCC2 : TCC Obligation

  % proof status :-)
  % xlat_pe_in_pt_range_TCC3 : TCC Obligation

  % proof status :-)
  % xlat_pe_in_pt_range_TCC4 : TCC Obligation

  % proof status :-)
  % xlat_pe_in_pt_range_TCC5 : TCC Obligation

  % proof status :-)
  xlat_pe_in_pt_range : Lemma
    Forall (addr : Memory_Address_4G, access : Memory_access,
              priv : Memory_privilege, s1, s2 : Linear_memory) :
      union(pm`ro_addr, pm`rw_addr)(addr) And
        is_linear_plain_memory?(pm) And
          pm`states(s1) And pm`states(s2) And
      OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)
                                   (PDBR)(s1))`base_addr, addr, access, priv)
           (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)))) And
        Not data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)
                                          (PDBR)(s1))`base_addr, addr, access, priv)
                 (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))`1
    Implies
      pe_in_pt_range?(s2, union(pm`ro_addr, pm`rw_addr), ptab_lvl)
        (xlat_idx(ptab_lvl, data(translate(pdir_lvl,
                    data(read_data(pm_phy, pdbr_data_type)
              (PDBR)(s1))`base_addr, addr, access, priv)
                (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))`2, addr))

  % proof status :-)
  linear_resolve_unchanged_pm_phy : Lemma
    Forall (addr : Memory_Address_4G, access : Memory_access) :
      union(pm`ro_addr, pm`rw_addr)(addr) And
        is_linear_plain_memory?(pm) And
          pm`states(s)
      Implies
        unchanged_memory_invariant?[Physical_memory](pm_phy`mem, pm`states,
          singleton(expr_2_super[Physical_memory, Address](
                      linear_resolve(addr, access))),
          difference(union(pm_phy`ro_addr, pm_phy`rw_addr),
                       address_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr))))

  % proof status :-)
  linear_resolve_read_transformers_ok : Lemma
    Forall (addr : Memory_Address_4G) :
```

```
    is_linear_plain_memory?(pm) And
      union(pm`ro_addr, pm`rw_addr)(addr)
    Implies
      transformers_ok?(pm`states, singleton(expr_2_super(
        linear_resolve(addr, Read))))

% proof status :-)
linear_resolve_write_transformers_ok : Lemma
  Forall (addr : Memory_Address_4G) :
    is_linear_plain_memory?(pm) And
      pm`rw_addr(addr)
    Implies
      transformers_ok?(pm`states, singleton(expr_2_super(
        linear_resolve(addr, Write))))

% proof status :-)
linear_resolve_transformer_invariant : Lemma
  Forall (addr : Memory_Address_4G, access : Memory_access) :
    is_linear_plain_memory?(pm) And
      union(pm`ro_addr, pm`rw_addr)(addr)
    Implies
      transformer_invariant?(pm`states, singleton(expr_2_super(
        linear_resolve(addr, access))))

% proof status :-)
% linear_resolve_states_TCC1 : TCC Obligation

% proof status :-)
linear_resolve_states : Lemma
  Forall (addr : Memory_Address_4G, access : Memory_access) :
    union(pm`ro_addr, pm`rw_addr)(addr) And
      is_linear_plain_memory?(pm) And
        pm`states(s) And
          OK?(linear_resolve(addr, access)(s))
  Implies
    pm_phy`states(state(linear_resolve(addr, access)(s)))

% proof status :-)
linear_resolve_same_result : Lemma
  Forall (addr : Memory_Address_4G, ac1, ac2 : Memory_access,
          s1, s2 : Linear_memory) :
    union(pm`ro_addr, pm`rw_addr)(addr) And
      is_linear_plain_memory?(pm) And
        pm`states(s1) And pm`states(s2) And
          OK?(linear_resolve(addr, ac1)(s1)) And
          OK?(linear_resolve(addr, ac2)(s2))
    Implies
      data(linear_resolve(addr, ac1)(s1)) =
        data(linear_resolve(addr, ac2)(s2))

% Translation of addresses within same page
%=========================================
```

```
% proof status :-)
delta_memory_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page)
      Implies
        offset(addr + delta) < max_linear_offset

% proof status :-)
% xlat_idx_same_page_address_TCC1 : TCC Obligation

% proof status :-)
% xlat_idx_same_page_address_TCC2 : TCC Obligation

% proof status :-)
xlat_idx_same_page_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
          base : {a : Memory_Address_4G |
                    aligned?(bits_per_level + pe_size)(offset(a))}) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page)
      Implies
        xlat_idx(lvl, base, addr) = xlat_idx(lvl, base, addr + delta)

% proof status :-)
% xlat_ofs_same_page_address_TCC1 : TCC Obligation

% proof status :-)
% xlat_ofs_same_page_address_TCC2 : TCC Obligation

% proof status :-)
xlat_ofs_same_page_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
          base : {a : Memory_Address_4G | aligned?(bus_width - (lvl + 1) *
                                                   bits_per_level)(offset(a))}) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page)
      Implies
        xlat_ofs(lvl, base, addr) + delta = xlat_ofs(lvl, base, addr + delta)

% proof status :-)
translate_same_page_address_ok : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
          base : {a : Memory_Address_4G |
                    aligned?(bits_per_level + pe_size)(offset(a))},
          access : Memory_access, priv : Memory_privilege) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
      OK?(translate(lvl, base, addr, access, priv)(s))
    Implies
      OK?(translate(lvl, base, addr + delta, access, priv)(s))

% proof status :-)
translate_same_page_address_result : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, lvl : Level,
```

```
          base : {a : Memory_Address_4G |
                    aligned?(bits_per_level + pe_size)(offset(a))},
          access : Memory_access, priv : Memory_privilege) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
      OK?(translate(lvl, base, addr, access, priv)(s))
    Implies
      translate(lvl, base, addr, access, priv)(s) =
        translate(lvl, base, addr + delta, access, priv)(s)

% proof status :-)
% linear_resolve_same_page_address_ok_TCC1 : TCC Obligation

% proof status :-)
linear_resolve_same_page_address_ok : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, ac : Memory_access) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
      OK?(linear_resolve(addr, ac)(s))
    Implies
      OK?(linear_resolve(addr + delta, ac)(s))

% proof status :-)
% linear_resolve_same_page_address_TCC1 : TCC Obligation

% proof status :-)
linear_resolve_same_page_address : Lemma
  Forall (addr : Memory_Address_4G, delta : nat, ac : Memory_access) :
    rem(expt(2, min_page))(offset(addr)) + delta < expt(2, min_page) And
      OK?(linear_resolve(addr, ac)(s))
    Implies
      data(linear_resolve(addr, ac)(s)) + delta =
        data(linear_resolve(addr + delta, ac)(s))

% Linear Memory Side Effects
%============================

% proof status :-)
address_block_split_type : Lemma
  Forall (a : Memory_Address_4G, bl : list[Byte]) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), union(pm`ro_addr, pm`rw_addr))
    Implies
      every(LAMBDA (t: [Address, list[Byte]]): Mem?(t`1`type_of) AND
                                                 0 <= t`1`offset And
                                                 t`1`offset < max_linear_offset)
        (split(min_page, a, bl))


% proof status :-)
% same_page_address_block_read_TCC1 : TCC Obligation

% proof status :-)
same_page_address_block_read : Lemma
```

```
    Forall (a1, a2 : Memory_Address_4G, s1, s2 : posnat) :
      is_linear_plain_memory?(pm) And
      pm`states(s) And
      union(pm`ro_addr, pm`rw_addr)(a2) And
      a1 <= a2 And a2 + s2 <= a1 + s1 And
      rem(expt(2, min_page))(offset(a2)) + s2 <= expt(2, min_page) And
      subset?(address_block(a1, s1), union(pm`ro_addr, pm`rw_addr))
    Implies
      subset?(address_block(data(linear_resolve(a2, Read)(s)), s2),
              union(pm_phy`ro_addr, pm_phy`rw_addr))

  % proof status :-)
  % same_page_address_block_write_TCC1 : TCC Obligation

  % proof status :-)
  same_page_address_block_write : Lemma
    Forall (a1, a2 : Memory_Address_4G, s1, s2 : posnat) :
      is_linear_plain_memory?(pm) And
      pm`states(s) And
      pm`rw_addr(a2) And
      a1 <= a2 And a2 + s2 <= a1 + s1 And
      rem(expt(2, min_page))(offset(a2)) + s2 <= expt(2, min_page) And
      subset?(address_block(a1, s1), pm`rw_addr)
    Implies
      subset?(address_block(data(linear_resolve(a2, Write)(s)), s2),
              pm_phy`rw_addr)

  % proof status :-)
  % split_linear_read_side_effects_states_TCC1 : TCC Obligation

  % proof status :-)
  split_linear_read_side_effects_states : Lemma
    Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
      is_linear_plain_memory?(pm) And
        subset?(address_block(a, length(bl)), union(pm`ro_addr, pm`rw_addr))
      Implies
        every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
                transformer_invariant?(pm`states,
                  singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
              (split(min_page, a, bl))

  % proof status :-)
  % split_linear_write_side_effects_states_TCC1 : TCC Obligation

  % proof status :-)
  split_linear_write_side_effects_states : Lemma
    Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
      is_linear_plain_memory?(pm) And
        subset?(address_block(a, length(bl)), pm`rw_addr)
      Implies
        every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
                transformer_invariant?(pm`states,
```

```
            singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
        (split(min_page, a, bl))

% proof status :-)
split_linear_read_side_effects_ok : Lemma
  Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)),
              union(pm`ro_addr, pm`rw_addr)) And
        pm`states(s)
    Implies
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
              OK?(linear_read_side_effect_in_page(e)(s)))
          (split(min_page, a, bl))

% proof status :-)
split_linear_write_side_effects_ok : Lemma
  Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), pm`rw_addr) And
        pm`states(s)
    Implies
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
              OK?(linear_write_side_effect_in_page(e)(s)))
          (split(min_page, a, bl))

% proof status :-)
split_linear_read_side_effects_data : Lemma
  Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)),
              union(pm`ro_addr, pm`rw_addr)) And
        pm`states(s)
    Implies
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
              OK?(linear_read_side_effect_in_page(e)(s)) Implies
            data(linear_read_side_effect_in_page(e)(s)) = e`2)
          (split(min_page, a, bl))

% proof status :-)
split_linear_write_side_effects_data : Lemma
  Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
    is_linear_plain_memory?(pm) And
      subset?(address_block(a, length(bl)), pm`rw_addr) And
        pm`states(s)
    Implies
      every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
              OK?(linear_write_side_effect_in_page(e)(s)) Implies
                data(linear_write_side_effect_in_page(e)(s)) = e`2)
          (split(min_page, a, bl))

% proof status :-)
```

```
  split_linear_read_side_effects_unchanged_memory : Lemma
    Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
      is_linear_plain_memory?(pm) And
        subset?(address_block(a, length(bl)), union(pm`ro_addr, pm`rw_addr))
      Implies
        every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
                unchanged_memory_invariant?(pm_phy`mem, pm`states,
                  singleton(expr_2_super(linear_read_side_effect_in_page(e))),
                              union(pm_phy`ro_addr, pm_phy`rw_addr)))
              (split(min_page, a, bl))

  % proof status :-)
  split_linear_write_side_effects_unchanged_memory : Lemma
    Forall (a : Memory_Address_4G, bl : (cons?[Byte])) :
      is_linear_plain_memory?(pm) And
        subset?(address_block(a, length(bl)), pm`rw_addr)
      Implies
        every(Lambda (e : [Memory_Address_4G, list[Byte]]) :
                unchanged_memory_invariant?(pm_phy`mem, pm`states,
                  singleton(expr_2_super(linear_write_side_effect_in_page(e))),
                              union(pm_phy`ro_addr, pm_phy`rw_addr)))
              (split(min_page, a, bl))



  % linear_unchanged_invariant
  %============================

  % proof status :-)
  linear_unchanged_invariant : Lemma
    Forall (transformers : PRED[[Linear_memory -> SuperResult[Linear_memory]]],
            addresses : PRED[Memory_Address_4G]) :
      is_linear_plain_memory?(pm) And
        subset?(addresses, union(pm`ro_addr, pm`rw_addr)) And
      (Forall (s : (pm`states)) :
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    transformers,
          virt_to_phys_range(s, addresses, union(singleton(Read),
                                                 singleton(Execute)))))
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states, transformers, addresses)



  % simple helpers for unchanged lemmas
  %=====================================

  % proof status :-)
  % pm_phy_read_after_resolve_ok_TCC1 : TCC Obligation

  % proof status :-)
  pm_phy_read_after_resolve_ok : Lemma
```

```
    Forall (a1 : Address, a2 : Memory_Address_4G, ac : Memory_access) :
      OK?(linear_resolve(a2, ac)(s)) And
      plain_memory?(pm_phy) And
      pm_phy`states(state(linear_resolve(a2, ac)(s))) And
      union(pm_phy`ro_addr, pm_phy`rw_addr)(a1)
    Implies
      OK?(memory_read(pm_phy`mem)(a1)(state(linear_resolve(a2, ac)(s))))

  % proof status :-)
  pm_resolve_address : Lemma
    Forall (a : Memory_Address_4G) :
      is_linear_plain_memory?(pm) And
      pm`states(s) And
      virt_to_phys_range(s, union(pm`ro_addr, pm`rw_addr),
                         union(singleton(Read), singleton(Execute)))(a)
    Implies
      difference(union(pm_phy`ro_addr, pm_phy`rw_addr),
                 address_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr)))(a)

  % proof status :-)
  pm_resolve_address_write : Lemma
    Forall (a : Memory_Address_4G) :
      is_linear_plain_memory?(pm) And
      pm`states(s) And
      virt_to_phys_range(s, pm`rw_addr, singleton(Write))(a)
    Implies
      difference(union(pm_phy`ro_addr, pm_phy`rw_addr),
                 address_in_pt_range?(s, union(pm`ro_addr, pm`rw_addr)))(a)


End Linear_Memory_Properties


Linear_Memory_Blessing_Properties
                        [% the underlying physical memory
                         Physical_memory : Type,
                         (Importing Plain_Memory[Physical_memory])
                         pm_phy : Plain_Memory
                        ] : Theory
Begin

  Importing Linear_Memory_Properties[Physical_memory, pm_phy]

  pm : Var Plain_Memory[Linear_memory]

  % Blessing results
  %==================

  % proof status :-)
  linear_plain_transformers_ok : Lemma
    is_linear_plain_memory?(pm)
  Implies
```

```
    transformers_ok?(pm`states, union(union(
      memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
      memory_write_transformers(pm`mem, pm`rw_addr)), union(
      memory_read_side_effect_super_transformers(pm`mem,
                                                 union(pm`ro_addr, pm`rw_addr)),
      memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))))

  % proof status :-)
  linear_plain_unchanged_memory_invariant : Lemma
    is_linear_plain_memory?(pm)
  Implies
    unchanged_memory_invariant?(pm`mem, pm`states,
      union(union(pm`other_actions,
      memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr))),
      union(memory_read_side_effect_super_transformers(pm`mem,
                                                 union(pm`ro_addr, pm`rw_addr)),
            memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))),
                                                 union(pm`ro_addr, pm`rw_addr))

  % proof status :-)
  linear_plain_unchanged_memory_invariant_write : Lemma
    is_linear_plain_memory?(pm)
  Implies
    unchanged_memory_invariant?(pm`mem, pm`states,
      memory_write_transformers(pm`mem, pm`rw_addr), pm`ro_addr)


  % proof status :-)
  linear_plain_unchanged_memory_write_invariant : Lemma
    is_linear_plain_memory?(pm)
  Implies
    unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)

  % proof status :-)
  linear_plain_transformer_invariant : Lemma
    is_linear_plain_memory?(pm)
  Implies
    transformer_invariant?(pm`states, union(union(pm`other_actions,
      union(memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
            memory_write_transformers(pm`mem, pm`rw_addr))),
      union(memory_read_side_effect_super_transformers(pm`mem,
                                                 union(pm`ro_addr, pm`rw_addr)),
            memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))))

  % proof status :-)
  linear_plain_changed_memory_invariant : Lemma
    is_linear_plain_memory?(pm)
  Implies
    changed_memory_invariant?(pm`mem, pm`states, pm`rw_addr)

  % We must use expression transformers instead of super transformers
  % to match the list argument of the side effect transformer in
```

```
  % side_effect_content_unchanged against the transformer obtained
  % from the union of memory side effect transformers. Super
  % transformers give us only equality on result states.

  % proof status :-)
  linear_plain_read_side_effect_unchanged : Lemma
    is_linear_plain_memory?(pm)
  Implies
    side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                  memory_read_side_effect(pm`mem))

  % proof status :-)
  linear_plain_write_side_effect_unchanged : Lemma
    is_linear_plain_memory?(pm)
  Implies
    side_effect_content_unchanged(pm`rw_addr, pm`states,
                                  memory_write_side_effect(pm`mem))

   %%%%% Final Blessing Result %%%%%
   % proof status :-)
   linear_memory_plain_memory : Lemma
     is_linear_plain_memory?(pm)
       Implies plain_memory?(pm)

End Linear_Memory_Blessing_Properties
```

B.5 challenge-phymem.pvs

```
Phy_Mem_Properties[(IMPORTING IA32) min, max : Memory_Address] : Theory
% Properties of physical memory
Begin
  IMPORTING Memory_Physical_Memory[min, max], Memory_Change_2
  IMPORTING Physical_Memory_Properties[min, max]

  addr : Var Address
  s : Var Physical_memory

  % proof status :-)
  phy_mem_write_transformers_ok : Lemma
      transformers_ok?(fullset[Physical_memory],
        memory_write_transformers(phy_mem, in_memory(min, max)))

  % proof status :-)
  phy_mem_read_transformers_ok : Lemma
      transformers_ok?(fullset[Physical_memory],
        memory_read_transformers(phy_mem, in_memory(min, max)))

  % proof status :-)
  phy_mem_read_side_effects_transformers_ok : Lemma
      transformers_ok?(fullset[Physical_memory],
        memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max)))
```

```
  % proof status :-)
  phy_mem_write_side_effects_transformers_ok : Lemma
      transformers_ok?(fullset[Physical_memory],
        memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max)))

  % proof status :-)
  phy_mem_unchanged_memory_invariant_read_transformers : Lemma
    unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
     memory_read_transformers(phy_mem, in_memory(min, max)),
                                in_memory(min, max))

  % proof status :-)
  phy_mem_unchanged_memory_invariant_read_side_effect_transformers : Lemma
    unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
     memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max)),
                                in_memory(min, max))

  % proof status :-)
  phy_mem_unchanged_memory_invariant_write_side_effect_transformers : Lemma
    unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
     memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max)),
                                in_memory(min, max))

  % proof status :-)
  phy_mem_unchanged_memory_write_invariant : Lemma
    unchanged_memory_write_invariant?(phy_mem, fullset[Physical_memory],
      in_memory(min, max))

  % proof status :-)
  phy_mem_changed_memory_invariant : Lemma
    changed_memory_invariant?(phy_mem, fullset[Physical_memory],
                                in_memory(min, max))

End Phy_Mem_Properties


Phy_Mem_Blessing[(IMPORTING IA32) min, max : Memory_Address] : Theory
% all physical memory is blessed memory
Begin
  Importing Phy_Mem_Properties[min, max], Plain_Memory

  oact : Var PRED[[Physical_memory->SuperResult[Physical_memory]]]

  phy_pm(oact) : Plain_Memory[Physical_memory] = (#
    mem := phy_mem,
    states := fullset[Physical_memory],
    ro_addr := emptyset[Address],
    rw_addr := in_memory(min, max),
    other_actions := oact
  #)
```

```
  % proof status :-)
  phy_mem_plain_memory : Lemma
    unchanged_memory_invariant?(phy_mem,
                  fullset[Physical_memory], oact, in_memory(min, max))
      Implies
        plain_memory?(phy_pm(oact))

End Phy_Mem_Blessing


Phy_Mem_Blessing_Data[Data : Type, (IMPORTING IA32) min, max : Memory_Address] : Theory
Begin

  Importing Phy_Mem_Blessing[min, max], Abstract_Read_Write

  addr : Var Address
  idt : Var (interpreted_data_type?[Data])

  % proof status :-)
  in_memory_blessed_memory : Lemma
    in_memory?(idt, addr, min, max) Implies
      in_blessed_memory?(idt, addr, in_memory(min, max))

End Phy_Mem_Blessing_Data


Phy_Mem_Challenge_Read_Same
  [Data : Type, (IMPORTING IA32) min, max : Memory_Address] : Theory
%
% read after write at the same place returns the item written
Begin

  IMPORTING Phy_Mem_Blessing[min, max],
        Plain_Mem_Properties_2[Physical_memory, Data],
        Phy_Mem_Blessing_Data[Data, min, max]

  dt : Var (interpreted_data_type?[Data])
  s : Var Physical_memory
  addr : Var Address
  data : Var Data


  % proof status :-)
  read_write_ok : Lemma in_memory?(dt, addr, min, max)
    IMPLIES
        OK?((
                write_data(phy_mem,dt)(addr, data) ##
                read_data(phy_mem,dt)(addr)
            )(s)
           )

  % proof status :-)
```

```
  % read_write_res_TCC1 : TCC Obligation

  % proof status :-)
  read_write_res : Lemma in_memory?(dt, addr, min, max)
    IMPLIES
    data((
                write_data(phy_mem,dt)(addr, data) ##
                read_data(phy_mem,dt)(addr)
              )(s))
      = data

end Phy_Mem_Challenge_Read_Same


Phy_Mem_Challenge_Read_Other
  [(IMPORTING IA32) min, max : Memory_Address, Data1, Data2 : Type] : Theory
% Phy_Mem_Challenge_Read_Other : Theory
%
% for read after write at a different place you can discard the write
%
Begin
  IMPORTING Phy_Mem_Blessing[min, max],
        Plain_Mem_Properties_3[Physical_memory, Data1, Data2],
        Phy_Mem_Blessing_Data

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  s : Var Physical_memory
  addr1, addr2 : Var Address
  data1 : Var Data1

  % proof status :-)
  read_write_other_ok : Lemma
    in_memory?(dt1, addr1, min, max) AND
    in_memory?(dt2, addr2, min, max) AND
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
    valid_in_mem(phy_mem,dt2)(addr2)(s)
      IMPLIES
        OK?
           ((
                write_data(phy_mem,dt1)(addr1, data1) ##
                read_data(phy_mem,dt2)(addr2)
            )(s)
           )

  % proof status :-)
  % read_write_other_res_TCC1 : TCC Obligation

  % proof status :-)
  % read_write_other_res_TCC2 : TCC Obligation

  % proof status :-)
```

```
  read_write_other_res : Lemma
    in_memory?(dt1, addr1, min, max) AND
    in_memory?(dt2, addr2, min, max) AND
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
    valid_in_mem(phy_mem,dt2)(addr2)(s)
      IMPLIES
        data((
                    write_data(phy_mem,dt1)(addr1, data1) ##
                    read_data(phy_mem,dt2)(addr2)
                  )(s))
          =
        data(read_data(phy_mem,dt2)(addr2)(s))


  % proof status :-)
  read_read_ok : Lemma
    in_memory?(dt1, addr1, min, max) AND
    in_memory?(dt2, addr2, min, max) AND
    valid_in_mem(phy_mem,dt1)(addr1)(s) And
    valid_in_mem(phy_mem,dt2)(addr2)(s)
      Implies
        OK?(
              ( read_data(phy_mem, dt1)(addr1) ##
                read_data(phy_mem, dt2)(addr2)
              )(s)
            )

  % proof status :-)
  % read_read_TCC1 : TCC Obligation

  % proof status :-)
  % read_read_TCC2 : TCC Obligation

  % proof status :-)
  read_read : Lemma
    in_memory?(dt1, addr1, min, max) AND
    in_memory?(dt2, addr2, min, max) AND
    valid_in_mem(phy_mem,dt1)(addr1)(s) And
    valid_in_mem(phy_mem,dt2)(addr2)(s)
      Implies
        data (
                    ( read_data(phy_mem, dt1)(addr1) ##
                      read_data(phy_mem, dt2)(addr2)
                    )(s)
                  )
        = data(read_data(phy_mem, dt2)(addr2)(s))

End Phy_Mem_Challenge_Read_Other
```

B.6 challenge-ptab-sync-master.pvs

```
Challenge_Ptab_Sync_Master : Theory
Begin

  Importing ptab_sync_master

% to do: specification and proofs

End Challenge_Ptab_Sync_Master
```

B.7 constants.pvs

```
% general constants and types
% (including architecture dependent ones)

General : Theory
% architecture independent constants and types
Begin
  Importing Number_Props

  bits_per_byte : posnat

  min_bits_per_byte : nat = 8

  bits_per_byte_minimum : Axiom bits_per_byte >= min_bits_per_byte

  max_byte : nat = expt(2, bits_per_byte)

  % proof status :-)
  % Byte_TCC1 : TCC Obligation

  Byte : Nonempty_Type = below(max_byte) containing 0

  % proof status :-)
  less_than_max_byte : Lemma Forall(n : nat) :
    n < expt(2, 8) Implies n < max_byte

End General


Register_Id : Datatype
Begin
  Mem_ : Mem?

  EAX_ : EAX?
  EBX_ : EBX?
  ECX_ : ECX?
  EDX_ : EDX?
  ESI_ : ESI?
  EDI_ : EDI?
  ESP_ : ESP?
  EBP_ : EBP?
```

```
  CS_ : CS?
  DS_ : DS?
  ES_ : ES?
  FS_ : FS?
  GS_ : GS?
  SS_ : SS?

  IP_ : IP?

  EFLAGS_ : EFLAGS?

  CR0_ : CR0?
  CR2_ : CR2?
  PDBR_ : PDBR?
  CR4_ : CR4?

  IDTR_ : IDTR?
  GDTR_ : GDTR?
  LDTR_ : LDTR?
  TR_ : TR?

  SYSENTER_CS_ : SYSENTER_CS?
  SYSENTER_ESP_ : SYSENTER_ESP?
  SYSENTER_EIP_ : SYSENTER_EIP?

End Register_Id


Address_Helpers : Theory
Begin

  Importing General, Register_Id

  Address : Type = [# type_of : Register_Id, offset : int #]

  % Address Constructors
  Mem(offset : int) : Address = (# type_of := Mem_, offset := offset #)
  CR0 : Address = (# type_of := CR0_, offset := 0 #)
  CR2 : Address = (# type_of := CR2_, offset := 0 #)
  CR4 : Address = (# type_of := CR4_, offset := 0 #)
  PDBR : Address = (# type_of:= PDBR_, offset := 0 #)

  CS : Address = (# type_of:= CS_, offset := 0 #)
  DS : Address = (# type_of:= DS_, offset := 0 #)
  ES : Address = (# type_of:= ES_, offset := 0 #)
  FS : Address = (# type_of:= FS_, offset := 0 #)
  GS : Address = (# type_of:= GS_, offset := 0 #)
  SS : Address = (# type_of:= SS_, offset := 0 #)

  Segment_Register_Address : Type =
    {a : Address | a = CS Or a = DS Or a = ES Or a = FS Or a = GS Or a = SS}
```

```
Memory_Address : Type = {a : Address | Mem?(a`type_of)}

a : Var Address
size : Var int

+(a, size) : Address = a With [(offset) := a`offset + size]

m_addr : Var Memory_Address

% proof status :-)
type_add_rewrite : Lemma
  (m_addr + size)`type_of = m_addr`type_of

% proof status :-)
offset_add_rewrite : Lemma
  (m_addr + size)`offset = m_addr`offset + size

% proof status :-)
type_add_memory : Judgement +(m_addr, size) Has_Type Memory_Address

% proof status :-)
address_add_zero : Lemma
  Forall (a : Address) : a + 0 = a

% proof status :-)
address_add_sum : Lemma
  Forall (a : Address, i, j : int) : (a + i) + j = a + (i + j)

a1, a2 : Var Address

< (a1, a2) : bool = a1`type_of = a2`type_of And a1`offset < a2`offset;

<= (a1, a2) : bool = a1`type_of = a2`type_of And a1`offset <= a2`offset

% Some register groups

GP?(id: Register_Id) : bool =
  EAX?(id) Or EBX?(id) Or ECX?(id) Or EDX?(id) Or ESI?(id) Or
    EDI?(id) Or EBP?(id) Or ESP?(id)

Segment_Reg?(id : Register_Id) : bool =
  CS?(id) Or DS?(id) Or ES?(id) Or FS?(id) Or GS?(id) Or SS?(id)

Control_Reg?(id : Register_Id) : bool =
  CR0?(id) Or CR2?(id) Or CR4?(id) Or PDBR?(id)

Table_Reg?(id : Register_Id) : bool =
  IDTR?(id) Or GDTR?(id) Or LDTR?(id) Or TR?(id)

MSR?(id : Register_Id) : bool =
  SYSENTER_CS?(id) Or SYSENTER_ESP?(id) Or SYSENTER_EIP?(id)
```

```
  % Register sizes

  % proof status :-)
  % reg_size_TCC1 : TCC Obligation

  % proof status :-)
  % reg_size_TCC2 : TCC Obligation

  reg_size(max : Memory_Address)(id : Register_Id) : Address =
    Cases id Of
      Mem_ : (# type_of := id, offset := offset(max) #),
      IP_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
      EFLAGS_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),

      IDTR_ : (# type_of := id, offset := expt(2, bits_per_byte * 8) #),
      GDTR_ : (# type_of := id, offset := expt(2, bits_per_byte * 8) #),
      LDTR_ : (# type_of := id, offset := expt(2, bits_per_byte * 14) #),
      TR_ : (# type_of := id, offset := expt(2, bits_per_byte * 14) #),

      SYSENTER_CS_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
      SYSENTER_ESP_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
      SYSENTER_EIP_ : (# type_of := id, offset := expt(2, bits_per_byte * 4) #)
    Else
      Cond
        GP?(id) -> (# type_of := id, offset := expt(2, bits_per_byte * 4) #),
        Segment_Reg?(id) -> (# type_of := id, offset := expt(2, bits_per_byte * 14) #),
        Control_Reg?(id) -> (# type_of := id, offset := expt(2, bits_per_byte * 4) #)
      EndCond
   EndCases

  reg_base(min : Memory_Address)(id : Register_Id) : Address =
    If Mem?(id) Then
      min
    Else
      (# type_of := id, offset := 0 #)
    Endif

  in_memory(min, max : Memory_Address)(a : Address) : bool =
    reg_base(min)(a`type_of) <= a And a < reg_size(max)(a`type_of)

  floor(b : posnat)(a : Address) : Address =
    a With [(offset) := floor(b)(offset(a))]

End Address_Helpers


IA32 : Theory
% architecture dependent constants and types
Begin
  IMPORTING Address_Helpers
```

```
  % type of addresses as subtype of natural numbers
  bus_width : nat = 32

  max_linear_offset : nat = expt(2, bus_width)
  max_linear : Address = Mem(max_linear_offset)

  min_linear : Address = Mem(0)

  % proof status :-)
  max_linear_offset_val : Lemma max_linear_offset = 4294967296

% max_linear_expt_val : Lemma offset(max_linear) = expt(2, 32)


  % memory addresses below 2^32, for those who need it
  Memory_Address_4G : Type = {a : Memory_Address | 0 <= a`offset And a`offset < max_linear_offset}

  % proof status :-)
  % mem_addr_4g_TCC1 : TCC Obligation

  mem_addr_4g(R : PRED[(in_memory(min_linear, max_linear))])(a : Memory_Address_4G) : bool = R(a)

  % smallest page size
  min_page : posnat = 12

  AUTO_REWRITE- bus_width, max_linear, max_linear_offset_val

End IA32
```

B.8 conversions.pvs

```
% $Id: conversions.pvs,v 1.11.4.2.2.1.2.3 2008/05/14 14:24:58 tews Exp $
%
% Author: Tjark Weber
% (c) 2007 Radboud University
%
% A shallow embedding of (some) C++ conversions as state transformers.

% Section numbers in comments throughout this file refer to
% "Programming languages - C++", ISO/IEC 14882:1998(E)

% Deviations from the C++ standard:
%
% This is preliminary work. Most of the standard hasn't even been considered
% yet.

% A note on PVS conversions: PVS has conversions as well. Should we declare
% the C++ conversion functions as PVS conversions, so that they are applied
% automatically, based on type information? We have decided against doing so
% for the following reasons. 1) Explicit conversions make the PVS translation
% of C++ code less readable, but the translated code is of little interest.
% What really matters is the proof state, which necessarily contains the
```

```
% conversions anyway. 2) Explicit conversions give us more control. It is not
% quite clear (to me, anyway) how the PVS type checker decides which
% conversions must be applied, and in which order. 3) Some C++ conversions
% require additional information (e.g. the lvalue-to-rvalue conversion must
% know the type of the stored value), and must therefore be made explicit
% anyway.

% Naming convention: conversions between different types are named
% <source>2<dest>
% where <source> and <dest> are the source and destination type, respectively, as
% given in types.pvs. Examples: bool2int, int2uint.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.1: Lvalue to rvalue conversion
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

LvalueToRvalueConversion[State, Data : Type] : THEORY
BEGIN

  IMPORTING Abstract_Read_Write_Plain

  pm : VAR Plain_Memory[State]
  dt : VAR (interpreted_data_type?[Data])

  s : VAR State

  a_ex : VAR [State -> ExprResult[State, Address]]

  % MV>> do we need l2r for non-fundamental types??
  % 4.1.2: l2r(class) returns an rvalue to a copy initialised temporary
  l2r(pm, dt)(a_ex) : [State -> ExprResult[State, Data]] =
  (
    a_ex ## lambda (a : Address):
    read_data(pm, dt)(a) ## lambda (d : Data):
    ok_result(d)
  )

  %MV>>
  % l2r_class(pm, copy_constructor)(a_ex) : ET[State, Data]
  % with_new_stackvar()(lambda (temporary : Address) :
  % copy_constructor(temporary, a_ex)
  % XXX

  %MV>>
  % It seems like we could restrict Data in ET[.., Data] to fundamental types;
  % For classes it seems the standard requires an in place creation or temporaries and the
  % only thing to do with a class is to access a member.

END LvalueToRvalueConversion
```

```
% MV:>> make promotions and conversions polymorphic over the fundamental types


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.5: Integral promotions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IntegralPromotions[State : Type] : THEORY
BEGIN

  IMPORTING Cpp_Types, Ok_Result_Rewrite

  % 4.5(1)

  % "An rvalue of type char [...] can be converted to an rvalue of type int if
  % int can represent all the values of the source type; otherwise, the source
  % rvalue can be converted to an rvalue of type unsigned int."

  % We take this to mean that in the former case, the value is not changed.

  % TODO: I have no idea how to model this correctly, for the *type* of the
  % conversion would depend on the condition. Also, I'm not sure what
  % should happen to source rvalues when int cannot represent all the
  % values of the source type, and unsigned int cannot represent the
  % particular rvalue. Below, only the first alternative (conversion to
  % int) is modeled. However, 4.7(3) specifies the behavior of conversions
  % to int when those conversions are *not* available as integral
  % promotions. To sum up, we can *always* convert to int; the behavior is
  % specified either by 4.5 or by 4.7. Likewise, we can always convert to
  % unsigned int; this is modeled in theory IntegralConversions below.

  % char2int
  % schar2int
  % uchar2int
  % short2int
  % ushort2int

  typ : Var Cpp_Subtype(cv({t : Cpp_Type | char?(t) Or schar?(t) Or uchar?(t) Or
                                           short?(t) Or ushort?(t)}))

  % proof status :-)
  % int_conversion_TCC1 : TCC Obligation

  % proof status :-)
  % int_conversion_TCC2 : TCC Obligation

  % proof status :-)
  % int_conversion_TCC3 : TCC Obligation

  int_conversion(typ) : [(range(typ)) -> (range(int))]
```

```
% proof status :-)
% int_conversion_spec_TCC1 : TCC Obligation

int_conversion_spec : AXIOM
  forall (n : (range(typ))) :
    % 4.5(1) / 4.7(3)
    range(int)(n) Implies int_conversion(typ)(n) = n
    % 4.7(3): otherwise conversion is allowed, but the value is
    % implementation-defined

int_conversion(typ)(n_ex : ET[State, (range(typ))]) : ET[State, (range(int))] =
(
  n_ex ## lambda (n : (range(typ))) :
  ok_result(int_conversion(typ)(n))
)

i_typ : Var Cpp_Subtype(cv(non_bool_integral_enum?))

% range to PVS int
range_to_int(i_typ)(n_ex : ET[State, (range(i_typ))]) : ET[State, int] =
(
  n_ex ## lambda (n : (range(i_typ))) :
  ok_result(n)
)

% PVS int to range
int_to_range(i_typ)(n_ex : ET[State, int]) : ET[State, (range(i_typ))] =
(
  n_ex ## lambda (n : int) :
  If range(i_typ)(n) Then
    ok_result(n)
  Else
    fatal_result
  Endif
)

% 4.5(2): TODO (wchar_t, enumeration types)

% 4.5(3): TODO (bit-fields)

% 4.5(4)

bool_ex : VAR ET[State, bool]

% proof status :-)
% bool2int_TCC1 : TCC Obligation

% proof status :-)
% bool2int_TCC2 : TCC Obligation

bool2int(bool_ex) : ET[State, (range(int))] =
```

```
  (
    bool_ex ## lambda (b : bool) :
    ok_result(if b then 1 else 0 endif)
  )

END IntegralPromotions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.7: Integral conversions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IntegralConversions[State : Type] : THEORY
BEGIN

  IMPORTING Cpp_Types, Ok_Result_Rewrite

  % 4.7(1): "An rvalue of an integer type can be converted to an rvalue of
  % another integer type."

  % The integer types are bool, char, wchar_t, (signed/unsigned) char,
  % (signed/unsigned) short, (signed/unsigned) int, (signed/unsigned) long
  % (3.9.1(7)).

  % bool_ex : VAR [State -> ExprResult[State, Semantics_bool]]
  % char_ex : VAR [State -> ExprResult[State, Semantics_char]]
  % wchar_t_ex : VAR [State -> ExprResult[State, Semantics_wchar_t]]
  % schar_ex : VAR [State -> ExprResult[State, Semantics_schar]]
  % uchar_ex : VAR [State -> ExprResult[State, Semantics_uchar]]
  % short_ex : VAR [State -> ExprResult[State, Semantics_short]]
  % ushort_ex : VAR [State -> ExprResult[State, Semantics_ushort]]
  % int_ex : VAR [State -> ExprResult[State, Semantics_int]]
  % uint_ex : VAR [State -> ExprResult[State, Semantics_uint]]
  % long_ex : VAR [State -> ExprResult[State, Semantics_long]]
  % ulong_ex : VAR [State -> ExprResult[State, Semantics_ulong]]

  %%%%%%%%%%
  % Conversions to bool: see 4.12
  %%%%%%%%%%

  %%%%%%%%%%
  % Conversions to char
  %%%%%%%%%%

  % bool2char(bool_ex) : [State -> ExprResult[State, Semantics_char]] =
  % (
  % bool_ex ## lambda (b : Semantics_bool):
  % ok_result(if b then 1 else 0 endif)
  % )
```

```
% TODO: wchar_t2char
% TODO: schar2char
% TODO: uchar2char
% TODO: short2char
% TODO: ushort2char
% TODO: int2char
% TODO: uint2char
% TODO: long2char
% TODO: ulong2char

%%%%%%%%%%
% Conversions to wchar_t
%%%%%%%%%%

% TODO

%%%%%%%%%%
% Conversions to schar
%%%%%%%%%%

% TODO

%%%%%%%%%%
% Conversions to uchar
%%%%%%%%%%

% TODO

%%%%%%%%%%
% Conversions to short
%%%%%%%%%%

% TODO

%%%%%%%%%%
% Conversions to ushort
%%%%%%%%%%

% TODO

%%%%%%%%%%
% Conversions to int
%%%%%%%%%%

% 4.7(5): "The conversions allowed as integral promotions are excluded from the
% set of integral conversions." Theory IntegralPromotions should therefore
% contain most of the necessary definitions already.

% TODO: long2int
% TODO: ulong2int

%%%%%%%%%%
```

```
  % Conversions to uint
  %%%%%%%%%%

  % bool2uint(bool_ex) : [State -> ExprResult[State, Semantics_uint]] =
  % (
  % bool_ex ## lambda (b : Semantics_bool):
  % ok_result(if b then 1 else 0 endif)
  % )

  % TODO: wchar_t2uint
  % TODO: schar2uint
  % TODO: uchar2uint
  % TODO: short2uint
  % TODO: ushort2uint

  % int2uint(int_ex) : [State -> ExprResult[State, Semantics_uint]] =
  % (
  % int_ex ## lambda (i : Semantics_int):
  % ok_result(mod(i, max_value(uint))) % 4.7(2)
  % )

  % TODO: long2uint
  % TODO: ulong2uint

  %%%%%%%%%%
  % Conversions to long
  %%%%%%%%%%

  % TODO

  %%%%%%%%%%
  % Conversions to ulong
  %%%%%%%%%%

  % TODO

END IntegralConversions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 4.12: Boolean conversions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

BooleanConversions[State : Type] : THEORY
BEGIN

  IMPORTING Abstract_Read_Write_Plain, Cpp_Types

  s : VAR State
```

```
  % TODO: there should probably be separate conversions for every arithmetic
  % and pointer type

  integral_to_bool(i_typ : Cpp_Subtype(cv(non_bool_integral_enum?)))
                   (n_ex : ET[State, (range(i_typ))]) : ET[State, bool] =
  (
    n_ex ## lambda (n : (range(i_typ))) :
    ok_result(not (n = 0))
  )

  % To do: floating_point_to_bool

  % proof status :-)
  % pointer_to_bool_TCC1 : TCC Obligation

  pointer_to_bool(p_typ : Cpp_Subtype(cv(pointer?)))
                  (p_ex : ET[State, (range_pointer(cv_base(p_typ)))]) : ET[State, bool] =
  (
    p_ex ## lambda (ptr_val : (range_pointer(cv_base(p_typ)))) :
    ok_result(not_null?(p_typ)(ptr_val))
  )

END BooleanConversions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% TODO: other conversions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Conversions: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Conversions[State : Type] : THEORY
BEGIN

  IMPORTING LvalueToRvalueConversion,
            IntegralPromotions,
            IntegralConversions,
            BooleanConversions

END Conversions
```

B.9 cpp-examples.pvs

```
% $Id: cpp-examples.pvs,v 1.26.6.2.2.4.2.2 2008/05/14 14:24:58 tews Exp $
%
```

```
% Author: Tjark Weber, Marcus Völp
% (c) 2007 Radboud University
%
% Some (rather simple) C++ programs and ((not yet) automatic) verification
% proofs.

Cpp_Examples[State : Type] : THEORY
BEGIN

  IMPORTING Cpp_Verification[State]

  STATES : PRED[State]

  % skip
  % proof status :-)
  spec01 : LEMMA
    Valid(STATES,
      skip[State],
    STATES)

  pm : Plain_Memory[State]

  a, b, c, n, m : Address

  % proof status :-)
  % PRECONDITION_TCC1 : TCC Obligation

  % proof status :-)
  % PRECONDITION_TCC2 : TCC Obligation

  % proof status :-)
  % PRECONDITION_TCC3 : TCC Obligation

  % proof status :-)
  % PRECONDITION_TCC4 : TCC Obligation

  PRECONDITION : PRED[State] =
    lambda (s : State):
      pm`states(s) and
        in_blessed_memory?(dt(int), n, pm`rw_addr) and
        in_blessed_memory?(dt(int), m, pm`rw_addr) and
        in_blessed_memory?(dt_bool, c, pm`rw_addr) and
        in_blessed_memory?(dt(char), a, pm`rw_addr) and
        in_blessed_memory?(dt(char), b, pm`rw_addr) And
        blocks_disjoint?(a, size_of(char), b, size_of(char)) And
        blocks_disjoint?(b, size_of(char), a, size_of(char)) And
        blocks_disjoint?(n, size_of(int), m, size_of(int)) And
        blocks_disjoint?(m, size_of(int), n, size_of(int))

  % proof status :-)
  % POSTCONDITION_TCC1 : TCC Obligation
```

```
POSTCONDITION : PRED[State] =
  lambda (s : State):
    let ss = read_data(pm, dt(int))(n)(s) in
      OK?(ss) and data(ss) = 42

% proof status :-)
% spec02_TCC1 : TCC Obligation

% a single assignment - partial correctness
% proof status :-)
spec02 : LEMMA
  plain_memory?(pm) implies
    valid(PRECONDITION,
      e2s(assign(pm, dt(int))(id(n), literal(42))),
    POSTCONDITION)

% a single assignment - total correctness
% proof status :-)
spec03 : LEMMA
  plain_memory?(pm) implies
    Valid(PRECONDITION,
      e2s(assign(pm, dt(int))(id(n), literal(42))),
    POSTCONDITION)

% No disjointness assumption should be necessary for this lemma.
% MV: Tjark: to remove the disjointness assumption we need
% sizeof(char) = 1 and something like:
% a /= b And sizeof(dt) = 1 => block_disjoint(a, b) And block_disjoint(b, a)
% Do we really need to check this automatically?

% proof status :-)
% spec04_TCC1 : TCC Obligation

% proof status :-)
% spec04_TCC2 : TCC Obligation

% two char assignments
% proof status :-)
spec04 : LEMMA
  plain_memory?(pm) implies
  valid(PRECONDITION,
    e2s(assign(pm, dt(char))(id(a), literal(42))) ##
    e2s(assign(pm, dt(char))(id(b), literal(42))),
    Lambda (s : State) : OK?(read_data(pm, dt(char))(a)(s)) And
      data(read_data(pm, dt(char))(a)(s)) = 42)

% if-else
% proof status :-)
spec05: LEMMA
  plain_memory?(pm) implies
  valid(PRECONDITION,
    e2s(assign(pm, dt_bool)(id(c), literal(true))) ##
```

```
      if_else(l2r(pm, dt_bool)(id(c)),
        e2s(assign(pm, dt(int))(id(n), literal(42))),
        skip),
    POSTCONDITION)

  % proof status :-)
  % spec06_TCC1 : TCC Obligation

  % proof status :-)
  % spec06_TCC2 : TCC Obligation

  % proof status :-)
  % spec06_TCC3 : TCC Obligation

  % proof status :-)
  % spec06_TCC4 : TCC Obligation

  % a switch statement (with several assignments)
  % proof status :-)
  spec06 : LEMMA
    plain_memory?(pm) implies
      valid(PRECONDITION,
        e2s(assign(pm, dt(char))(id(a), literal(0))) ##
        e2s(assign(pm, dt(char))(id(b), literal(1))) ##
        switch(l2r(pm, dt(char))(id(a)), (: 0, 1 :),
          case(0) ##
            skip ##
          case(1) ##
            e2s(assign(pm, dt(int))(id(n), literal(42))) ##
            break ##
          default ##
            e2s(assign(pm, dt(int))(id(n), literal(0)))
        ),
      POSTCONDITION)


  % a trivial infinite loop
  % proof status :-)
  spec07: LEMMA
    valid(PRECONDITION,
      while(literal(true),
        skip),
    lambda (s : State): false)


  % a nontrivial loop

  % proof status :-)
  % N_TCC1 : TCC Obligation

  N : {n : (range(int)) | n >= 0} = 2
```

```
  % proof status :-)
  % range_int_values_TCC1 : TCC Obligation

  % proof status :-)
  range_int_values : Lemma
    Forall (i : int) :
      -32767 <= i And i <= 32767
    Implies
      range(int)(i)

  % proof status :-)
  % spec08_TCC1 : TCC Obligation

  % proof status :-)
  spec08: LEMMA
    plain_memory?(pm) implies
    valid(PRECONDITION,
      e2s(assign(pm, dt(int))(id(n), literal(0))) ##
      e2s(assign(pm, dt(int))(id(m), literal(0))) ##
      while(N + 1, lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
            e2s(postinc(pm, int)(id(n))) ##
            e2s(postinc(pm, int)(id(m)))),
      lambda (s : State) :
        OK?(read_data(pm, dt(int))(m)(s)) Implies
          data(read_data(pm, dt(int))(m)(s)) = N)

  % proof status :-)
  spec09 : Lemma
    plain_memory?(pm) implies
      valid(PRECONDITION,
        e2s(assign(pm, dt(int))(id(m), literal(0))) ##
        for(
          N + 1,
          e2s(assign(pm, dt(int))(id(n), literal(0))),
          lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
          postinc(pm, int)(id(n)),
          e2s(postinc(pm, int)(id(m)))),
        Lambda (s : State) :
          OK?(read_data(pm, dt(int))(m)(s)) Implies
            data(read_data(pm, dt(int))(m)(s)) = N)

  % proof status :-)
  % spec10_TCC1 : TCC Obligation

%>>
  % proof status :-( unfinished
  spec10 : Lemma
    plain_memory?(pm) implies
      valid(PRECONDITION,
        for(
          e2s(assign(pm, dt(int))(id(n), literal(0))),
          lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
```

```
        postinc(pm, int)(id(n)),
        skip)(
        % P
        (Lambda (s0 : State) : Lambda (res : StmtResult[State]) :
           OK?(res) And OK?(read_data(pm, dt(int))(n)(state(res))) And
             0 <= data(read_data(pm, dt(int))(n)(state(res))) And
             data(read_data(pm, dt(int))(n)(state(res))) < N
        ),
        % Q
        (Lambda (s0 : State) : Lambda (res : StmtResult[State]) :
           OK?(res) And OK?(read_data(pm, dt(int))(n)(state(res))) And
             data(read_data(pm, dt(int))(n)(state(res))) = N
        ),
        % R
        (Lambda (s0 : State) : Lambda (res : StmtResult[State]) :
           OK?(res) And OK?(read_data(pm, dt(int))(n)(state(res))) And
           data(read_data(pm, dt(int))(n)(state(res))) <= N
        ),
        % variant
        (Lambda (s0 : State) : Lambda (s : State) :
           If OK?(read_data(pm, dt(int))(n)(s)) And
              data(read_data(pm, dt(int))(n)(s)) <= N
           Then
             N - data(read_data(pm, dt(int))(n)(s))
           Else
             N
           Endif),
        % order
        Lambda (i, j : int) : i < j
        ),
      Lambda (s : State) :
        OK?(read_data(pm, dt(int))(n)(s)) Implies
          data(read_data(pm, dt(int))(n)(s)) = N)

END Cpp_Examples

% Rewrite: Hoare rules
% (auto-rewrite "valid" "Valid" "PRECONDITION" "POSTCONDITION")
% (smash)
% (skosimp* : preds? t)

% Required Axioms:
% size_of(typ)
% const and volatile models


% (auto-rewrite
% "address_of_spec" "address_of_spec2" "add_simplification"
% "address_add_zero" "address_add_sum"
% "add_spec" "add_spec2" "same_array_spec" "same_array_spec2"
% "mod_range" "size_of" "uidt" "check_bounds_spec"
% "er_plus_ax" "er_minus_ax" "er_div_ax" "er_times_ax" "er_neg_ax"
```

```
% "extended_real_superset_of_number_fields")

% Rewrite: Expressions
% (auto-rewrite
% "literal" "id" "member" "postinc" "postdec" "deref" "arrow"
% "unary_minus" "unary_minus_unsigned" "unary_not" "unary_bitnot"
% "preinc" "predec" "UnaryExpressions.sizeof" "ptm_member" "ptm_arrow"
% "arrow" "times" "div" "div_float" "mod" "plus" "plus_ptr"
% "postinc_ptr" "preinc_ptr" "subscript" "minus" "minus_ptr"
% "postdec_ptr" "predec_ptr" "minus_ptr_ptr" "cmp" "lt" "gt" "le" "ge"
% "lt_ptr" "gt_ptr" "le_ptr" "ge_ptr"
% "cmp_pointer" "eq_bool" "eq" "not_equal" "not_equal_bool" "eq_ptr"
% "not_equal_ptr" "bitand" "bitxor" "bitor" "and_exp" "or_exp"
% "conditional_expr" "assign" "assign_times" "assign_div" "assign_mod"
% "assign_div_float" "assign_plus" "assign_plus_ptr" "assign_minus"
% "assign_minus_ptr" "assign_bitand" "assign_bitxor" "assign_bitor"
% "comma")

% Rewrites: Plain Memory
% (auto-rewrite
% "plain_memory_write_ok_single"
% "plain_memory_read_data_ok"
% "plain_memory_write_ok_q_expr"
% "plain_memory_write_ok_q_stmt"
% "in_blessed_memory_rw_ro"
% "pm_q_prop_ok"
% "pm_q_prop_single_read"
% "pm_q_prop_single_write"
% "pm_q_prop_read_write_single"
% "pm_q_prop_read_write_other_single"
% "pm_q_prop_read_read_single"
% "plain_memory_read_write_other_single_data"
% "plain_memory_read_read"
% "plain_memory_read_write_res"
% "pm_q_prop_read_ok_expr"
% "pm_q_prop_read_ok_stmt"
% "pm_q_prop_read_pm_q_prop_expr"
% "pm_q_prop_read_pm_q_prop_stmt"
% "pm_q_prop_read_write_q_expr"
% "pm_q_prop_read_write_q_stmt"
% "plain_memory_read_write_q_data_expr"
% "plain_memory_read_write_q_data_stmt"
% "pm_q_prop_read_write_other_q_expr"
% "pm_q_prop_read_write_other_q_stmt"
% "pm_q_prop_read_read_q_expr"
% "pm_q_prop_read_read_q_stmt"
% "plain_memory_read_write_other_q_data_expr"
% "plain_memory_read_write_other_q_data_stmt"
% "plain_memory_read_read_q_data_expr"
% "plain_memory_read_read_q_data_stmt"
% "pm_q_prop_ok_result_single"
% "pm_q_prop_ok_result_q_expr"
```

```
% "pm_q_prop_ok_result_q_stmt"
% "pm_q_prop_read_ok_result_single"
% "pm_q_prop_read_ok_result_q_expr"
% "pm_q_prop_read_ok_result_q_stmt"
% "pm_q_prop_e2s"
% "pm_q_prop_stmt_e2s"
% "pm_q_prop_read_e2s"
% "pm_q_prop_read_stmt_e2s"
% "pm_q_prop_read_ok__pm_q_prop_read_expr"
% "pm_q_prop_read_ok__pm_q_prop_read_stmt"
% )

% Rewrite: Switch
% (auto-rewrite
% "if_else"
% "switch"
% "member"
% "break_catch_break"
% "break_break_stmt"
% "break_break_catch_continue"
% "break_break_catch_default"
% "break_break_lift_stmt"
% "break_break_case"
% "break_break_default"
% "break_break"
% "stmt_break_break"
% "break_stmt_break"
% "break_stmt_catch_continue"
% "break_stmt_catch_default"
% "break_stmt_lift"
% "break_stmt_case"
% "break_stmt_default"
% "continue_catch_continue"
% "continue_break_stmt"
% "continue_break_catch_continue"
% "continue_break_catch_default"
% "continue_break_lift_stmt"
% "continue_break_case"
% "continue_break_default"
% "continue_continue"
% "stmt_continue_continue"
% "continue_stmt_continue"
% "continue_stmt_catch_continue"
% "continue_stmt_catch_default"
% "continue_stmt_lift"
% "continue_stmt_case"
% "continue_stmt_default"
% "switch_stmt"
% "stmt_switch_stmt"
% "switch_case_taken"
% "stmt_switch_case_taken"
% "switch_case_not_taken"
```

```
% "stmt_switch_case_not_taken"
% "switch_default"
% "stmt_switch_default"
% "switch_catch_break"
% "stmt_switch_catch_break"
% "switch_catch_continue"
% "stmt_switch_catch_continue"
% "switch_catch_default"
% "stmt_switch_catch_default"
% "default_stmt"
% "stmt_default_stmt"
% "default_case"
% "stmt_default_case"
% "default_default"
% "stmt_default_default"
% "default_catch_break"
% "stmt_default_catch_break"
% "default_catch_continue"
% "stmt_default_catch_continue"
% "default_catch_default"
% "stmt_default_catch_default"
% "return_void_result"
% "return_ex_result"
% "stmt_remove_catch_default"
% "stmt_remove_case"
% "stmt_remove_default"
% "stmt_remove_catch_continue"
% "stmt_remove_catch_break"
% "break_break_stmt_expr"
% "break_break_catch_continue_expr"
% "break_break_catch_default_expr"
% "break_break_lift_stmt_expr"
% "break_break_case_expr"
% "break_break_default_expr"
% "continue_break_stmt_expr"
% "continue_break_catch_continue_expr"
% "continue_break_catch_default_expr"
% "continue_break_lift_stmt_expr"
% "continue_break_case_expr"
% "continue_break_default_expr"
% )

% Rewrite: Skip / E2S
% (auto-rewrite
% "skip_ok"
% "skip_state"
% "skip_elemination"
% "stmt_skip_elemination"
% "stmt_ok_lift"
% "stmt_ok_lift_expr"
% "e2s_ok"
% "stmt_e2s_ok"
```

```
% "e2s_state"
% "stmt_e2s_state"
% "ok_result_fexpr"
% "ok_result_fstmt"
% "e2s_merge"
% "stmt_e2s_merge"
% "e2s_expr"
% "stmt_e2s_expr"
% )


% Rewrite: Loops by unrolling
% (auto-rewrite
% "while_unroll"
% "stmt_while_unroll"
% "iterate_while_ok"
% "while_hang"
% "while_unroll_expr"
% "while_unroll_lexpr"
% "stmt_while_unroll_expr"
% "stmt_while_unroll_lexpr"
% "while_unroll_0"
% "do_while_unroll"
% "do_while_inv_unroll"
% "for_unroll"
% "for_inv_unroll"
% "while_to_while_no_cb"
% "while_invariant?!"
% "while_variant?!"
% "while_inv_rewrite_termination_result!"
% "while_inv_rewrite_data_ok!"
% "while_inv_rewrite_data_break!"
% "while_inv_rewrite_data_return!"
% "stmt_while_inv_rewrite_termination_result!"
% "stmt_while_inv_rewrite_data_ok!"
% "stmt_while_inv_rewrite_data_break!"
% "stmt_while_inv_rewrite_data_return!"
% "pm_q_prop_while!"
% "pm_q_prop_stmt_while!"
% )

% Rewrite: Composition
% (auto-rewrite
% "composition_assoc_rewrite_stmt_ok"
% "forward_s_c_s_of_s_expr"
% "comp_eval_if_ok_fstmt"
% "stmt_eval_if_ok_fstmt"
% "comp_eval_if_ok_fstmt_cstmt"
% "stmt_eval_if_ok_fstmt_cstmt"
% "composition_assoc_rewrite_1"
% "composition_assoc_rewrite_2"
```

```
% "composition_assoc_rewrite_3"
% "composition_assoc_rewrite_4"
% "composition_assoc_rewrite_5"
% "composition_assoc_rewrite_6"
% "composition_assoc_rewrite_7"
% "composition_assoc_rewrite_8"
% "composition_assoc_rewrite_expr_1"
% "composition_assoc_rewrite_expr_2"
% "composition_assoc_rewrite_expr_3"
% "composition_assoc_rewrite_expr_4"
% "composition_assoc_rewrite_expr_5"
% "composition_assoc_rewrite_expr_6"
% "composition_assoc_rewrite_stmt_ok"
% "composition_assoc_rewrite_stmt_ok_expr"
% "composition_assoc_rewrite_stmt_ok_expr_lexpr"
% "comp_eval_if_ok_fstmt"
% "comp_eval_if_ok_fstmt_expr"
% "comp_eval_if_ok_fstmt_cstmt_expr"
% "stmt_eval_if_ok_fstmt_expr"
% "stmt_eval_if_ok_fstmt_cstmt_expr"
% "comp_eval_if_ok_fexpr_expr"
% "comp_eval_if_ok_fexpr"
% "stmt_eval_if_ok_fexpr"
% "expr_eval_if_ok_fexpr"
% "composition_assoc_expression_rewrite"
% "composition_assoc_expression_rewrite_stmt"
% "stmt_eval_if_ok_fexpr_expr"
% "expr_eval_if_ok_fexpr_expr"
% "ok_result_fexpr"
% "ok_result_fstmt"
% "composition_assoc_rewrite_stmt_ok_cstmt"
% )

% Rewrite: Allocation
% (auto-rewrite
% "allocate_stack" "deallocate_stack" "with_new_stackvar"
% )

% Rewrite: Ok Result
% (auto-rewrite
% "ok_result_ok"
% "ok_result_data"
% "ok_result_q_ok"
% "ok_result_q_state"
% "ok_result_q_data"
% "ok_result_elemination_1"
% "ok_result_elemination_2"
% "ok_result_state"
% )

% Rewrite: Conversions
% (auto-rewrite
```

```
% "l2r" "integral_to_bool" "pointer_to_bool" "bool2int"
% )

% Rewrite: Precond rules
% (auto-rewrite
% "comp_simple_expr_simple_expr_ok"
% "comp_simple_expr_simple_expr_data"
% "comp_simple_expr_simple_expr_state"
% "comp_simple_stmt_simple_expr_ok"
% "comp_simple_stmt_simple_expr_data"
% "comp_simple_stmt_simple_expr_state"
% )
```

B.10 cpp-verification.pvs

```
% $Id: cpp-verification.pvs,v 1.2.6.1.4.1 2008/05/13 14:47:26 tews Exp $
%
% Author: Tjark Weber
% (c) 2007 Radboud University
%
% Verification environment for C++. Main theory, imports all other theories
% that are needed to verify C++ programs in PVS.

Cpp_Verification[State : Type] : THEORY
BEGIN

  Importing Expressions, Conversions, Hoare, Statement_Rewrites

  pm : Plain_Memory[State]

END Cpp_Verification
```

B.11 datatype_model.pvs

```
% $Id: datatype_model.pvs,v 1.1.6.2.4.2 2008/05/14 14:24:58 tews Exp $
%
% Author: Tjark Weber
% (c) 2007 Radboud University
%
% A definition of two PODs with different "type tags", where
% read_data(pm, pod_2)(addr) after write_data(pm, pod_1)(addr) fails.
%
% This is merely an example.
%
% The contents of this file should probably be moved to "builtin_models.pvs"
% when the latter typechecks again.

% Note that there is (yet another) problem with the current POD formalization:
% it does not guarantee that PODs can be copied via char arrays.

% Two trivial PODs, both with unit semantics.
```

```
Datatype_Model : THEORY
BEGIN

  IMPORTING Unit, Interpreted_Data[Unit]

  % proof status :-)
  % uidt_1_TCC1 : TCC Obligation

  uidt_1 : (uninterpreted_data_type?) =
    (# size := 1,
       valid? := LAMBDA (bl: list[Byte], a : Address):
                   bl = (: 0 :)
    #)

  % proof status :-)
  % pod_1_TCC1 : TCC Obligation

  % proof status :-)
  % pod_1_TCC2 : TCC Obligation

  % proof status :-)
  % pod_1_TCC3 : TCC Obligation

  % proof status :-)
  % pod_1_TCC4 : TCC Obligation

  pod_1 : (pod_data_type?) =
    (# uidt := uidt_1,
       to_byte := LAMBDA (d: Unit, a : Address):
                    (: 0 :),
       to_mask := Lambda (d : Unit, a : Address) :
                    zero_mask?(size(uidt_1)),
       from_byte := LAMBDA (bl: list[Byte], a : Address):
                      if bl = (: 0 :) then up(unit) else bottom endif
    #)

  % proof status :-)
  % uidt_2_TCC1 : TCC Obligation

  uidt_2 : (uninterpreted_data_type?) =
    (# size := 1,
       valid? := LAMBDA (bl: list[Byte], a : Address):
                   bl = (: 1 :)
    #)

  % proof status :-)
  % pod_2_TCC1 : TCC Obligation

  % proof status :-)
  % pod_2_TCC2 : TCC Obligation
```

```
  % proof status :-)
  % pod_2_TCC3 : TCC Obligation

  pod_2 : (pod_data_type?) =
    (# uidt := uidt_2,
       to_byte := LAMBDA (d: Unit, a : Address):
                    (: 1 :),
       to_mask := Lambda (d : Unit, a : Address) :
                    zero_mask?(size(uidt_2)),
       from_byte := LAMBDA (bl: list[Byte], a : Address):
                      if bl = (: 1 :) then up(unit) else bottom endif
    #)

END Datatype_Model

Read_After_Write_Fails[State : Type] : THEORY
BEGIN

  IMPORTING Datatype_Model, Statements, Plain_Mem_Properties[State]

  pm : VAR Plain_Memory[State]
  s : VAR State
  a : VAR Address

  % proof status :-)
  % plain_mem_write_list_states_TCC1 : TCC Obligation

  % proof status :-)
  plain_mem_write_list_states : LEMMA
    Forall(size : nat, bl : list[Byte]) :
      size = length(bl) and
      plain_memory?(pm) and
      pm`states(s) and
      subset?(address_block(a, size), pm`rw_addr) and
      OK?(memory_write_list(pm`mem)(a, bl)(s)) implies
        pm`states(state(memory_write_list(pm`mem)(a, bl)(s)))

  % proof status :-)
  plain_mem_write_list_read_list_ok : LEMMA
    Forall(size : nat, bl : list[Byte]) :
      size = length(bl) and
      plain_memory?(pm) and
      pm`states(s) and
      subset?(address_block(a, size), pm`rw_addr) and
      OK?(memory_write_list(pm`mem)(a, bl)(s)) implies
      OK?(memory_read_list(pm`mem)(a, size)
           (state(memory_write_list(pm`mem) (a, bl)(s))))

  % proof status :-)
  Fatal_Result : LEMMA
    plain_memory?(pm) and
    pm`states(s) and
```

```
    in_blessed_memory?(pod_1, a, pm`rw_addr) implies
      Fatal?( (write_data(pm, pod_1)(a, unit) ##
                read_data(pm, pod_2)(a))(s) )

END Read_After_Write_Fails


% The same for two perhaps more interesting PODs: bool and Byte.

Datatype_Model_2 : THEORY
BEGIN

  IMPORTING Interpreted_Data[bool]

  % proof status :-)
  % uidt_bool_TCC1 : TCC Obligation

  uidt_bool : (uninterpreted_data_type?) =
    (# size := 2,
       valid? := LAMBDA (bl: list[Byte], a : Address):
                   bl = (: 0, 0 :) or bl = (: 0, 1 :)
    #)

  % proof status :-)
  % pod_bool_TCC1 : TCC Obligation

  % proof status :-)
  % pod_bool_TCC2 : TCC Obligation

  % proof status :-)
  % pod_bool_TCC3 : TCC Obligation

  % proof status :-)
  % pod_bool_TCC4 : TCC Obligation

  % proof status :-)
  % pod_bool_TCC5 : TCC Obligation

  % proof status :-)
  % pod_bool_TCC6 : TCC Obligation

  pod_bool : (pod_data_type?) =
    (# uidt := uidt_bool,
       to_byte := LAMBDA (d: bool, a : Address):
                    if d then (: 0, 1 :) else (: 0, 0 :) endif,
       to_mask := Lambda (d : bool, a : Address) :
                    zero_mask?(size(uidt_bool)),
       from_byte := LAMBDA (bl: list[Byte], a : Address):
                      if bl = (: 0, 1 :) then up(true)
                      else if bl = (: 0, 0 :) then up(false)
                      else bottom endif endif
    #)
```

```
  IMPORTING Interpreted_Data[Byte]

  % proof status :-)
  % uidt_Byte_TCC1 : TCC Obligation

  % proof status :-)
  % uidt_Byte_TCC2 : TCC Obligation

  uidt_Byte : (uninterpreted_data_type?) =
    (# size := 2,
       valid? := LAMBDA (bl: list[Byte], a : Address):
                   length(bl) = 2 and car(bl) = 1
    #)

  % proof status :-)
  % pod_Byte_TCC1 : TCC Obligation

  % proof status :-)
  % pod_Byte_TCC2 : TCC Obligation

  % proof status :-)
  % pod_Byte_TCC3 : TCC Obligation

  % proof status :-)
  % pod_Byte_TCC4 : TCC Obligation

  % proof status :-)
  % pod_Byte_TCC5 : TCC Obligation

  pod_Byte : (pod_data_type?[Byte]) =
    (# uidt := uidt_Byte,
       to_byte := LAMBDA (d: Byte, a : Address):
                    (: 1, d :),
       to_mask := Lambda (d : Byte, a : Address) :
                    zero_mask?(size(uidt_Byte)),
       from_byte := LAMBDA (bl: list[Byte], a : Address):
                      if length(bl) = 2 and car(bl) = 1 then up(car(cdr(bl))) else bottom endif
    #)

END Datatype_Model_2


Read_After_Write_Fails_2[State : Type] : THEORY
BEGIN

  IMPORTING Datatype_Model_2, Statements, Plain_Mem_Properties[State]

  pm : VAR Plain_Memory[State]
  s : VAR State
  a : VAR Address
```

```
  % proof status :-)
  plain_mem_write_list_states : LEMMA
    Forall(size : nat, bl : list[Byte]) :
      size = length(bl) and
      plain_memory?(pm) and
      pm`states(s) and
      subset?(address_block(a, size), pm`rw_addr) and
      OK?(memory_write_list(pm`mem)(a, bl)(s)) implies
        pm`states(state(memory_write_list(pm`mem)(a, bl)(s)))

  % proof status :-)
  plain_mem_write_list_read_list_ok : LEMMA
    Forall(size : nat, bl : list[Byte]) :
      size = length(bl) and
      plain_memory?(pm) and
      pm`states(s) and
      subset?(address_block(a, size), pm`rw_addr) and
      OK?(memory_write_list(pm`mem)(a, bl)(s)) implies
      OK?(memory_read_list(pm`mem)(a, size)
           (state(memory_write_list(pm`mem) (a, bl)(s))))

  % proof status :-)
  Fatal_Result : LEMMA
    plain_memory?(pm) and
    pm`states(s) and
    in_blessed_memory?(pod_bool, a, pm`rw_addr) implies
      Fatal?( (write_data(pm, pod_bool)(a, true) ##
                read_data(pm, pod_Byte)(a))(s) )

END Read_After_Write_Fails_2
```

B.12 device_memory.pvs

```
Expand_State[State, T : Type] : Theory
Begin

  Importing State_Transformer

  Expand_state : Type = [#
    state : State,
    expansion : T
  #]

  em_lift(stf : [State -> SuperResult[State]]) :
                              [Expand_state -> SuperResult[Expand_state]] =
    Lambda (s : Expand_state) :
      Let res = stf(s`state) In
        Cases res Of
          OK(state) : OK(s With [(state) := state]),
          Abnormal(state) : Abnormal(s With [(state) := state]),
          Bottom : Bottom
```

```
        EndCases

  em_lift(P : PRED[State]) : PRED[Expand_state] =
    Lambda (em : Expand_state) : P(em`state)

  em_lift(P : PRED[[State -> SuperResult[State]]])
          (q : [Expand_state -> SuperResult[Expand_state]]) : bool =
    Exists (p : (P)) : q = em_lift(p)

  states : Var PRED[State]
  transformers : Var PRED[[State -> SuperResult[State]]]

  % proof status :-)
  em_transformers_ok : Lemma
    transformers_ok?(states, transformers) Implies
      transformers_ok?(em_lift(states), em_lift(transformers))

  % proof status :-)
  em_transformer_invariant : Lemma
    transformer_invariant?(states, transformers) Implies
      transformer_invariant?(em_lift(states), em_lift(transformers))

  % proof status :-)
  em_lift_singleton : Lemma
    Forall (q : [State -> SuperResult[State]]) :
      singleton(em_lift(q)) = em_lift(singleton(q))

End Expand_State

Expand_State2[State, T, Data : Type] : Theory
Begin

  Importing Expand_State[State, T]

  em_lift(stf : [State -> ExprResult[State, Data]]) :
                              [Expand_state -> ExprResult[Expand_state, Data]] =
    Lambda (s : Expand_state) :
      Let res = stf(s`state) In
        Cases res Of
          OK(state, data) : OK(s With [(state) := state], data),
          Exception(ex_type, state) : Exception(ex_type, s With [(state) := state]),
          Fatal : Fatal,
          Hang : Hang
        EndCases

  % proof status :-)
  em_lift_expr_2_super : Lemma
    Forall (stf : [State -> ExprResult[State, Data]]) :
      expr_2_super(em_lift(stf)) = em_lift(expr_2_super(stf))

End Expand_State2
```

```
Expand_State3[State, T : Type] : Theory
Begin

  Importing Expand_State2, Plain_Memory

  pm : Var Memory_struct[State]
  em_pm : Var Memory_struct[Expand_state[State, T]]

  states : Var PRED[State]
  addresses : Var PRED[Address]
  transformers : Var PRED[[State -> SuperResult[State]]]

  % proof status :-)
  em_memory_read_transformers : Lemma
    (em_pm`memory_read =
       Lambda (a : Address) : em_lift[State, T, Byte](pm`memory_read(a)))
  Implies
    memory_read_transformers(em_pm, addresses) =
      em_lift(memory_read_transformers(pm, addresses))

  % proof status :-)
  em_memory_write_transformers : Lemma
    (em_pm`memory_write =
       Lambda (a : Address, b : Byte) :
         em_lift[State, T, Unit](pm`memory_write(a, b)))
  Implies
    memory_write_transformers(em_pm, addresses) =
      em_lift(memory_write_transformers(pm, addresses))

  % proof status :-)
  em_unchanged_memory_invariant : Lemma
    (em_pm`memory_read = Lambda (a : Address) : em_lift[State, T, Byte](pm`memory_read(a))) And
    unchanged_memory_invariant?(pm, states, transformers, addresses)
  Implies
    unchanged_memory_invariant?(em_pm, em_lift(states),
                                em_lift(transformers), addresses)

  % proof status :-)
  em_changed_memory_invariant : Lemma
    (em_pm`memory_read =
       Lambda (a : Address) : em_lift[State, T, Byte](pm`memory_read(a))) And
    (em_pm`memory_write =
       Lambda (a : Address, b : Byte) :
         em_lift[State, T, Unit](pm`memory_write(a, b))) And
    changed_memory_invariant?(pm, states, addresses)
  Implies
    changed_memory_invariant?(em_pm, em_lift(states), addresses)

  % proof status :-)
  % em_const_unchanged_memory_invariant_TCC1 : TCC Obligation

  % proof status :-)
```

```
  em_const_unchanged_memory_invariant : Lemma
    Forall (q : [Expand_state[State, T] -> SuperResult[Expand_state[State, T]]]) :
      (em_pm`memory_read = Lambda (a : Address) : em_lift[State, T, Byte](pm`memory_read(a))) And
      (Forall (s : Expand_state[State, T]) : has_next_state(q(s)) Implies state(q(s))`state = s`state)
    Implies
      unchanged_memory_invariant?(em_pm, em_lift(states), singleton(q), addresses)

  se_transformer : Var
    [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

  % proof status :-)
  em_side_effect_content_unchanged : Lemma
    side_effect_content_unchanged(addresses, states, se_transformer)
  Implies
    side_effect_content_unchanged(addresses, em_lift(states),
      Lambda (a : Address, bl : list[Byte], cp : bool) :
        em_lift[State, T, list[Byte]](se_transformer(a, bl, cp)))

End Expand_State3


Device_Memory [Physical_memory : Type,
               (Importing Plain_Memory[Physical_memory])
               pm : Plain_Memory,
               Device_state : Type] : Theory
Begin

  Importing Expand_State3

  Device_memory : Type = Expand_state[Physical_memory, Device_state]

  s : Var Device_memory
  a : Var Address
  b : Var Byte
  size : Var nat

  device_read_side_effect : Var
    [Address, list[Byte], bool ->
                    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]
  device_write_side_effect : Var
    [Address, list[Byte], bool ->
                    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]

  device_pm(device_read_side_effect,
            device_write_side_effect) : Memory_struct[Device_memory] =
  (#
    memory_read := Lambda (a : Address) :
      em_lift[Physical_memory, Device_state, Byte](memory_read(pm`mem)(a)),
    memory_write := Lambda (a : Address, b : Byte) :
      em_lift[Physical_memory, Device_state, Unit](memory_write(pm`mem)(a, b)),
    memory_read_side_effect :=
      Lambda (a : Address, bl : list[Byte], cp : bool) :
```

```
      em_lift[Physical_memory, Device_state, list[Byte]](
        memory_read_side_effect(pm`mem)(a, bl, cp)) ## Lambda (bl1 : list[Byte]) :
      device_read_side_effect(a, bl1, cp),
    memory_write_side_effect :=
      Lambda (a : Address, bl : list[Byte], cp : bool) :
      em_lift[Physical_memory, Device_state, list[Byte]](
        memory_write_side_effect(pm`mem)(a, bl, cp)) ## Lambda (bl1 : list[Byte]) :
      device_write_side_effect(a, bl1, cp)
  #)

End Device_Memory

Challenge_Device_Memory [Physical_memory : Type,
                         (Importing Plain_Memory[Physical_memory])
                         pm_phy : Plain_Memory,
                         Device_state : Type] : Theory
Begin

  Importing Device_Memory[Physical_memory, pm_phy, Device_state],
            Plain_Mem_Rewrites

  device_read_side_effect : Var
    [Address, list[Byte], bool ->
                    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]
  device_write_side_effect : Var
    [Address, list[Byte], bool ->
                    [Device_memory -> ExprResult[Device_memory, list[Byte]]]]

  addresses : Var PRED[Address]
  pm : Var Plain_Memory[Device_memory]

  drse_super_transformers(addresses, device_read_side_effect) :
                          PRED[[Device_memory -> SuperResult[Device_memory]]] =
    Lambda (q : [Device_memory -> SuperResult[Device_memory]]) :
      Exists (a : Address, bl : list[Byte], cp : bool) :
        subset?(address_block(a, length(bl)), addresses) And
          q = expr_2_super(device_read_side_effect(a, bl, cp))

  dwse_super_transformers(addresses, device_write_side_effect) :
                          PRED[[Device_memory -> SuperResult[Device_memory]]] =
    Lambda (q : [Device_memory -> SuperResult[Device_memory]]) :
      Exists (a : Address, bl : list[Byte], cp : bool) :
        subset?(address_block(a, length(bl)), addresses) And
          q = expr_2_super(device_write_side_effect(a, bl, cp))



  is_device_plain_memory?(device_read_side_effect, device_write_side_effect)
                          (pm) : bool =
    % plain memory
    pm`mem = device_pm(device_read_side_effect, device_write_side_effect) And
    plain_memory?(pm_phy) And
```

```
    pm`states = em_lift(pm_phy`states) And
    subset?(pm`other_actions, em_lift(pm_phy`other_actions)) And
    subset?(pm`rw_addr, pm_phy`rw_addr) And
    subset?(pm`ro_addr, union(pm_phy`ro_addr, difference(pm_phy`rw_addr, pm`rw_addr))) And
    % side effect properties
    transformers_ok?(pm`states, union(
      drse_super_transformers(union(pm`ro_addr, pm`rw_addr),
                              device_read_side_effect),
      dwse_super_transformers(pm`rw_addr, device_write_side_effect))) And
    unchanged_memory_invariant?(pm`mem, pm`states, union(
      drse_super_transformers(union(pm`ro_addr, pm`rw_addr),
                              device_read_side_effect),
      dwse_super_transformers(pm`rw_addr, device_write_side_effect)),
                                union(pm`ro_addr, pm`rw_addr)) And
    side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                  device_read_side_effect) And
    side_effect_content_unchanged(pm`rw_addr, pm`states,
                                  device_write_side_effect)

  % Helper Lemmas
  %-----------------
  % proof status :-)
  pm_dev_plain : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    plain_memory?(pm_phy)

  % proof status :-)
  pm_dev_states : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    pm`states = em_lift(pm_phy`states)

  % proof status :-)
  pm_dev_read : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    memory_read(pm`mem) =
      Lambda (a : Address) :
        em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a))

  % proof status :-)
  pm_dev_write : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    memory_write(pm`mem) =
      Lambda (a : Address, b : Byte) :
        em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy`mem)(a, b))
```

```
% proof status :-)
pm_dev_read_side_effect : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)
Implies
  memory_read_side_effect(pm`mem) =
    Lambda (a : Address, bl : list[Byte], cp : bool) :
      em_lift[Physical_memory, Device_state, list[Byte]](
        memory_read_side_effect(pm_phy`mem)(a, bl, cp)) ## lambda (bl1 : list[Byte]) :
      device_read_side_effect(a, bl1, cp)

% proof status :-)
pm_dev_write_side_effect : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)
Implies
  memory_write_side_effect(pm`mem) =
    Lambda (a : Address, bl : list[Byte], cp : bool) :
      em_lift[Physical_memory, Device_state, list[Byte]](
        memory_write_side_effect(pm_phy`mem)(a, bl, cp)) ## lambda (bl1 : list[Byte]) :
      device_write_side_effect(a, bl1, cp)

% proof status :-)
pm_dev_ro_rw_addr : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)
Implies
  subset?(union(pm`ro_addr, pm`rw_addr), union(pm_phy`ro_addr, pm_phy`rw_addr))

% Plain Memory Properties
%-------------------------

% proof status :-)
device_plain_transformers_ok : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)
Implies
  transformers_ok?(pm`states, union(
    memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
    memory_write_transformers(pm`mem, pm`rw_addr)))

% proof status :-)
device_plain_unchanged_memory_invariant : Lemma
  is_device_plain_memory?(device_read_side_effect,
                          device_write_side_effect)(pm)
Implies
  unchanged_memory_invariant?(pm`mem, pm`states, union(union(pm`other_actions,
          memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr))),
    union(memory_read_side_effect_super_transformers(pm`mem,
                                                     union(pm`ro_addr, pm`rw_addr)),
          memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))),
```

```
                                                    union(pm`ro_addr, pm`rw_addr))

  % proof status :-)
  device_plain_unchanged_memory_invariant_write : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    unchanged_memory_invariant?(pm`mem, pm`states,
      memory_write_transformers(pm`mem, pm`rw_addr), pm`ro_addr)

  % proof status :-)
  device_plain_unchanged_memory_write_invariant : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)

  % proof status :-)
  device_plain_changed_memory_invariant : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    changed_memory_invariant?(pm`mem, pm`states, pm`rw_addr)

  % proof status :-)
  device_plain_read_side_effect_unchanged : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                  memory_read_side_effect(pm`mem))

  % proof status :-)
  device_plain_write_side_effect_unchanged : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    side_effect_content_unchanged(pm`rw_addr, pm`states,
                                  memory_write_side_effect(pm`mem))

  % proof status :-)
  device_plain_transformers_ok2 : Lemma
    is_device_plain_memory?(device_read_side_effect,
                            device_write_side_effect)(pm)
  Implies
    transformers_ok?(pm`states, union(
      memory_read_side_effect_super_transformers(pm`mem,
                                                 union(pm`ro_addr, pm`rw_addr)),
      memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr)))

  % Main Blessing Result
  %======================
```

```
  % proof status :-)
  device_memory_plain_memory : Lemma
    Forall (pm) :
      is_device_plain_memory?(device_read_side_effect,
                              device_write_side_effect)(pm)
        Implies plain_memory?(pm)

End Challenge_Device_Memory
```

B.13 everything.pvs

```
Import_All : Theory
% This theory imports (directly or indirectly) *every* theory
% of the project. This way one can conveniently rerun all proofs
% with prove-importchain (C-c C-p i) on this theory.
Begin
  % Please add all new theories here !!

  Importing Phy_Mem_Challenge_Read_Same, Phy_Mem_Challenge_Read_Other,
            Linear_Memory_Blessing_Properties,
            Statement_Rewrites,
            Allocation_Table, Challenge_Random_Device, Expressions

  % data type models
  Importing PDBR_Data_Model, PDE_Model, PTE_Model, PF_EC_Model, Address_Model

  % test theories
  Importing Rewrite_Test, Search_Example, Cpp_Examples

  Importing Read_After_Write_Fails, Read_After_Write_Fails_2

  % intermediate stuff, that will disappear later
  Importing PDBR_Datatype, PDE_Datatype, PTE_Datatype,
        Physical_Memory_Corollaries, Block_Disjoint_Rewrites,
        Posnat_induction, Set_Lift


End Import_All
```

B.14 expressions.pvs

```
% $Id: expressions.pvs,v 1.34.2.2.2.5.2.4 2008/05/14 14:24:58 tews Exp $
%
% Author: Tjark Weber (based on lecture material by Hendrik Tews)
% (c) 2007 Radboud University
%
% A shallow embedding of (some) C++ expressions as state transformers.

% Section numbers in comments throughout this file refer to
% "Programming languages - C++", ISO/IEC 14882:1998(E)
```

```
% Deviations from the C++ standard:
%=====================================
%
% Evaluation Order
%-------------------
% Note that the C++ standard leaves the evaluation order of subexpressions
% mostly unspecified. The behavior is undefined unless each expression
% satisfies a (somewhat obscure) side condition (cf. 5.4). We do not model
% this here; the following definitions instead use some fixed evaluation order.
% Michael Norrish in his paper
% "Deterministic expressions in C", 8th European Symposium on Programming
% (ETAPS), Amsterdam. 1999. LNCS 1576, pp147-161
% shows that the evaluation order is actually irrelevant for expressions that
% satisfy the side condition of 5.4. At the source code level, the evaluation
% order can be made deterministic by disallowing operands with side effects,
% storing their values in temporary variables instead: e.g.
% x = f() + g();
% could be replaced by
% t1 = f(); t2 = g(); x = t1 + t2;
%
% Modulo Arithmetic
%--------------------
% C++ defines some types and their corresponding operations to follow
% arithmetic modulo 2^n where n is the number of bits in the value
% representation. To be able detect integer overflows we do not model modulo
% arithmetic directly but instead we add the function mod which is defined to
% be the identity function for the value range of the respective datatype but
% which is otherwise undefined.
%
% Integer division "/" and remainder "%" (5.6.4) behave differently, see below.
%
% Bit-shift operators "<<" and ">>" (5.8.1) behave differently; see below.

%Open Issues:
%============
% - Calling Convention (5.2 functions)
% I don't think the C++ standard specifies just how exactly function
% arguments are stored on the stack (or even if there is a stack, for.that
% matter.) "call_f" obviously defines a certain (hopefully reasonable)
% behavior.
%
% This is defined in the ABI documents. (e.g., cdecl is defined in
% the calling convention is defined in 3.9 in the "System V ABI Intel 386
% Architecture Processor Supplement")
%
% - passing values to functions rather than addresses (5.2 functions)
% For specification/verification, it might be useful to have yet another PVS
% function for f which takes values (rather than expressions or addresses) as
% arguments. This hasn't been implemented yet.
%
% - entry points + address ranges for functions (5.2 functions)
% We should probably define for each function the range of addresses where
```

```
% the code of this function resides and the address of the entry point
%
% − should we implement exception values; some FPU ops and div 0 generate a
% well−defined hardware exception (e.g., Exception 0: divide error). Using
% Extended_Reals such an implementation could specify exception values (e.g.,
% x / 0 = div_0), check for these values and instead of exiting with Fatal
% return with the appropriate exception.

% To do:
%========
% − define expressions for bitfield types
% − with_new_refvar(ref, init)(Lambda (ref : (range(ref))) : xxx)
% − define << and >> operator
% − address of operator (&)
% − pointer assignment
% on assignment we have to adjust the pointer address to refer to the
% base class subobject of a derived object.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.1: Primary expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

PrimaryExpressions[State, Data : Type] : THEORY
BEGIN

  IMPORTING State_Transformer_Lift

  x : VAR Data

  % 5.1.2
%>> MV: use Cpp type; define literals for int, char, float, bool // string
  literal(x) : [State −> ExprResult[State, Data]] = ok_result(x)

  % 5.1.7
%>> MV: x is always an address (??)
  id(x) : [State −> ExprResult[State, Data]] = ok_result(x)

END PrimaryExpressions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.2: Postfix expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% A C++ function "ret_data f(arg_data a)" currently corresponds to two PVS
% functions
%
% f(a : Address) : [State −> StmtResult[State, Semantics_<ret_data>]] =
```

```
% body
%
% (where each branch in the body must be terminated by a "return" statement)
% and
%
% call_f(a : [State -> ExprResult[State, Semantics_<arg_data>]]) :
% [State -> ExprResult[State, Semantics_<ret_data>]] =
% with_new_stackvar( lambda (a_addr : Address):
% assign(a_addr, a) ##
% f(a_addr)) ##
% catch_return

PostfixExpressions[State : Type] : THEORY
BEGIN

  IMPORTING Abstract_Read_Write_Plain, Cpp_Types

  pm : VAR Plain_Memory[State]

  a_ex : VAR ET[State, Address]

  % 5.2.5: Class member access
  member(a_ex, (offset : nat)) : [State -> ExprResult[State, Address]] =
  (
    a_ex ## lambda (a : Address):
    ok_result(a + offset)
  )

  % 5.2.1: Subscripting
  % "The expression E1[E2] is identical (by definition to *((E1) + (E2))
  % see subscript in arithmetic expressions

  % 5.2.5 pt 3: -> operator
  % "the expression E1->E2 is converted to the equivalent from (*(E1)).E2
  % see arrow in unary expressions


  % proof status :-)
  % postinc_TCC1 : TCC Obligation

  % 5.2.6 pt 1: C++'s .++ operator
  postinc(pm, (typ : Cpp_Subtype(v(non_bool_integral?))))(a_ex) :
    ET[State, (range(typ))] =
  (
    a_ex ## lambda (a : Address):
    read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
    If range(typ)(mod(typ)(n + 1)) Then
      write_data(pm, dt(typ))(a, mod(typ)(n + 1)) ##
      ok_result(n)
    Else
      fatal_result
    Endif
```

```
  )

  % proof status :-)
  % postinc_float_TCC1 : TCC Obligation

  % proof status :-)
  % postinc_float_TCC2 : TCC Obligation

  % proof status :-)
  % postinc_float_TCC3 : TCC Obligation

%>> example of floating point operation
  postinc_float(pm, (typ : Cpp_Subtype(v(floating_point?))))(a_ex) :
    ET[State, (range_floating_point(cv_base(typ)))] =
  (
    a_ex ## lambda (a : Address):
    read_data(pm, dt_cv_float(typ))(a) ##
      lambda (n : (range_floating_point(cv_base(typ)))):
    If range_floating_point(cv_base(typ))(mod_float(cv_base(typ))(er_plus(n, 1))) Then
      write_data(pm, dt_cv_float(typ))(a, mod_float(cv_base(typ))(er_plus(n, 1))) ##
      ok_result(n)
    Else
      fatal_result
    Endif
  )

  % 5.2.6 pt 1: C++'s ++ operator for pointers to complete object
  % deferred to arithmetic expressions

  % 5.2.6 pt 1
  % .++ for argument type bool (deprecated)

  % 5.2.6 pt 2: C++'s .-- operator
  % .-- is not defined for argument type bool
  postdec(pm, (typ : Cpp_Subtype(v(non_bool_integral?))))(a_ex) :
    ET[State, (range(typ))] =
  (
    a_ex ## lambda (a : Address):
    read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
    If range(typ)(mod(typ)(n - 1)) Then
      write_data(pm, dt(typ))(a, mod(typ)(n - 1)) ##
      ok_result(n)
    Else
      fatal_result
    Endif
  )

  % To do: postdec_float

  % 5.2.6 pt 2: C++'s .-- operator for pointers to complete object
  % deferred to arithmetic expressions
```

```
  % 5.2.7 dynamic_cast
  %TODO: definition of dynamic_cast (5.2.7)

  % 5.2.8 typeid
  % TODO: definition of typeid (5.2.8)

  % 5.2.9 static_cast
  % TODO: definition of static_cast (5.2.9)
  % - static_cast<reference_type>(v) = lvalue otherwise = rvalue 5.2.9.1
  % - static_cast can only increase qualifiedness (+volatile / +const)

  % 5.2.10 reinterpret_cast
  % TODO: define reinterpret_cast (5.2.10)
  % - reinterpret_cast<reference_type>(v) = lvalue otherwise = rvalue 5.2.10.1
  % - reinterpret_cast<>(const v) = const 5.2.10.2
  % - mapping is implementation defined

  % MV: we cannot define reinterpret cast in a generic way using
  % from_byte(T2)(to_byte(T1)); (5.2.10.3) the representation may or may
  % not change !!!

  % 5.2.11 const_cast
  % TODO: define const_cast (5.2.11)

END PostfixExpressions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.3: Unary expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

UnaryExpressions[State : Type] : Theory
BEGIN

  IMPORTING PostfixExpressions

  pm : VAR Plain_Memory[State]

  a_ex : Var ET[State, Address]

  % proof status :-)
  % deref_TCC1 : TCC Obligation

  % 5.3.1: C++'s * operator
  deref(pm, (p_typ : Cpp_Subtype(cv(pointer?))))
        (p_ex : ET[State, (range_pointer(cv_base(p_typ)))]) :
       ET[State, Memory_Address] =
  (
    p_ex ## lambda (ptr_val : (range_pointer(cv_base(p_typ)))) :
    If not_null?(p_typ)(ptr_val) And
       check_bounds(p_typ)(ptr_val) And
```

```
      up?(address_of(p_typ)(ptr_val))
  Then
    ok_result(down(address_of(p_typ)(ptr_val)))
  Else
    fatal_result
  Endif
)

% To do: * for pointer to function

% proof status :-)
% arrow_TCC1 : TCC Obligation

% proof status :-)
% arrow_TCC2 : TCC Obligation

% 5.2.5 pt 3: -> operator
% "the expression E1->E2 is converted to the equivalent from (*(E1)).E2
arrow(pm, (p_typ : Cpp_Subtype(cv(Lambda (t : Cpp_Type) : pointer?(t) And
  (class?(typ(t)) Or union?(typ(t))))))
     (p_ex : ET[State, (range_pointer(cv_base(p_typ)))], offset : nat) :
    ET[State, Address] =
(
   deref(pm, p_typ)(p_ex) ## Lambda (a : Address) :
   member(ok_result(a), offset)
)

% 5.3.1 pt 2 C++'s & operator
% not needed for variables; = member() for class members

% 5.3.1 pt 6 unary + operator
% (not needed as it returns the value of the argument)

% proof status :-)
% unary_minus_TCC1 : TCC Obligation

% proof status :-)
% unary_minus_TCC2 : TCC Obligation

% 5.3.1 pt 7 unary - operator
unary_minus(typ : Cpp_Subtype(cv({t : Cpp_Type |
                (char?(t) Or wchar_t?(t) Or signed_integer?(t))})))
           (n_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
(
  n_ex ## lambda (n : (range(typ))) :
  If range(typ)(mod(typ)(-n)) Then
    ok_result(mod(typ)(-n))
  Else
    fatal_result
  Endif
)
```

```
  % To do: unary_minus_float

  % proof status :-)
  % unary_minus_unsigned_TCC1 : TCC Obligation

  % proof status :-)
  % unary_minus_unsigned_TCC2 : TCC Obligation

  % 5.3.1 pt 7 "the negative of an unsigned quantity is computed by
  % substracting its value from 2^n"
  unary_minus_unsigned(typ : Cpp_Subtype(cv(unsigned_integer?)))
              (n_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
  (
    n_ex ## lambda (n : (range(typ))) :
    If range(typ)(mod(typ)(2^max_value_bits(typ) - n)) Then
      ok_result(mod(typ)(2^max_value_bits(typ) - n))
    Else
      fatal_result
    Endif
  )

  % 5.3.1 pt 8: C++'s ! operator
  unary_not(typ : Cpp_Subtype(cv(bool?)))
            (b_ex : ET[State, bool]) : ET[State, bool] =
  (
    b_ex ## lambda (b : bool) :
    ok_result(Not b)
  )

% To do
  % 5.3.1 pt 9 C++'s ~ operator
  % "The result is the one's complement of its operand."
  %
  %>> MV and Tjark think (hope) that this means we apply one's complement
  % to the binary representation and see whether a valid value
  % comes out of it.
  unary_bitnot(typ : Cpp_Subtype(cv(non_bool_integral_enum?)))
                (n_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
  (
    n_ex ## lambda (n : (range(typ))) :
    % Because dt(typ) is a pod for all of these types we can take any address:
    % Mem(0)
    Let a = Mem(0),
        bitnot_value =
          from_byte(dt(typ))(
            apply_bitop_valuefield(value_bitmask(typ), binary_not)
                                  (to_byte(dt(typ))(n, a)), a)
    In
      If up?(bitnot_value) And range(typ)(down(bitnot_value)) Then
        ok_result(down(bitnot_value))
      Else
        fatal_result
```

```
    Endif
)


% proof status :-)
% preinc_TCC1 : TCC Obligation

% 5.3.2 pt 1: C++'s ++. operator
% Note that the prefix increment/decrement operators return an lvalue (i.e.
% an address).
preinc(pm, (typ : Cpp_Subtype(v(non_bool_integral?)))(a_ex) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
  If range(typ)(mod(typ)(n + 1)) Then
    write_data(pm, dt(typ))(a, mod(typ)(n + 1)) ##
    ok_result(a)
  Else
    fatal_result
  Endif
)

% To do: preinc_float

% 5.3.2 pt 1: C++'s ++. operator for pointers
% deferred to arithmetic expressions

% 5.3.2 pt 2: C++'s --. operator
% --. is not defined for argument type bool
predec(pm, (typ : Cpp_Subtype(v(non_bool_integral?)))(a_ex) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  read_data(pm, dt(typ))(a) ## lambda (n : (range(typ))):
  If range(typ)(mod(typ)(n - 1)) Then
    write_data(pm, dt(typ))(a, mod(typ)(n - 1)) ##
    ok_result(a)
  Else
    fatal_result
  Endif
)

% To do: predec_float

% 5.3.2 pt 1: C++'s --. operator for pointers
% deferred to arithmetic expressions

% proof status :-)
% sizeof_TCC1 : TCC Obligation

% 5.3.3 sizeof operator (only the type parameter form is implemented)
```

```
  % TODO: define the type std::size_t and return this type in sizeof
  % this is elsa's size_of_type expression
  %
  % 7.17 pt 2 C − Standard: size_t is an unsigned integer type
  sizeof(typ : Cpp_Type) : ET[State, (range(uint))] =
    If range(uint)(size_of(typ)) Then
      ok_result(size_of(typ))
    Else
      fatal_result
    Endif

  % 5.3.4 C++ new operator
  % TODO

  % 5.3.5 C++ delete operator
  % TODO

END UnaryExpressions

% 5.4 definition of cast (T)exp
% TODO

PointerToMemberExpressions[State : Type] : THEORY
BEGIN

  Importing UnaryExpressions

  pm : VAR Plain_Memory[State]

  a_ex : Var ET[State, Address]

  % 10 pt 5 "A base class subobject might have a layout different from the
  % layout of the most derived object of the same type."

  % 9.2 pt 17 "A pointer to a pod−struct object, suitably converted using a
  % reinterpret_cast, points to its initial member [...] and vice
  % versa."

  % GCC seems to adjust the addresses of a pointer upon pointer assignment. For
  % example, in the assignment B* bp = &a where class A : C, B {}; bp will be
  % set to point to the B subobject in A. This goes as far as that a vtable is
  % alloated for B. However, the vtable value is not identical to a
  % freestanding B.

%>> Können wir zeigen, das ein anderes Vorgehen Typinformation in Pointern oder
% in Pointer auf Klassenmember braucht?

%>> Ich würde daher pointer to member expressions wie folgt formalisieren:
  % We assume that members of a subobject are not reordered with members of
  % other subobjects or with members of the derived object. We can therefore
  % adjust the pointer to point to a base class. This base class may possibly
  % have a layout different from a standalone object.
```

```
  %
  % However, to detect this, type information must be stored with the object.
  % (??)
  %
  % We thus require the pointer to which pointer to member expressions are
  % bound to have the same class as the pointer to member.

  % proof status :-)
  % ptm_member_TCC1 : TCC Obligation

  % 5.5 pt 2 C++'s .* operator for pointer to member
  % To do:
  % 5.5 pt 4 "If the dynamic type of the object does not contain the member
  % to which the pointer refers, the behaviour is undefined."
  % range(typ) only contains offsets that are valid for the possibly derived
  % class typ(typ).
  ptm_member(typ : Cpp_Subtype(cv(pointer_to_member?)))
              (a_ex, (ptm_ex : ET[State, (range_ptm(cv_base(typ)))])) :
      ET[State, Address] =
  (
    ptm_ex ## Lambda (ofs : (range_ptm(cv_base(typ)))) :
      % the offset of the member
    member(a_ex, ofs)
  )

  % proof status :-)
  % ptm_arrow_TCC1 : TCC Obligation

  % 5.5 pt 3 C++'s ->* operator for pointer to member
  % To do:
  % check typ(ptr_typ) is a unambiguous accessible base class of typ(ptm_typ).
  ptm_arrow(pm, (ptm_typ : Cpp_Subtype(cv(pointer_to_member?))))
             (p_ex : ET[State, (range_pointer(cv_base(
                                                 pointer(typ(cv_base(ptm_typ))))))],
              ptm_ex : ET[State, (range_ptm(cv_base(ptm_typ)))]) :
      ET[State, Address] =
  (
    ptm_ex ## Lambda (ofs : (range_ptm(cv_base(ptm_typ)))) :
    arrow(pm, pointer(typ(cv_base(ptm_typ))))(p_ex, ofs)
  )

END PointerToMemberExpressions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.6: Multiplicative operators
% 5.7: Additive operators
% 5.8: Shift operators
% 5.9: Relational operators
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
ArithmeticExpressions[State : Type] : THEORY
BEGIN

  IMPORTING UnaryExpressions

  pm : Var Plain_Memory[State]

  a_ex : Var ET[State, Address]

  i_typ : Var Cpp_Subtype(cv(non_bool_integral_enum?))

  % The standard defines that bool values are subject to integral promotions:
  % 5 pt 9 "This pattern is called the 'usual arithmetic conversions' ...
  % Otherwise, the integral promotion (4.5) shall be performed."

  % Multiplicative Operators
  %============================

  % 5.6: C++'s * operator (multiplication)
  times(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
    ET[State, (range(i_typ))] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))) :
    n2_ex ## lambda (n2 : (range(i_typ))) :
    If range(i_typ)(mod(i_typ)(n1 * n2)) Then
      ok_result(mod(i_typ)(n1 * n2))
    Else
      fatal_result
    Endif
  )

  % To do: times_float

  % 5.6.4: "If the second operand of / or % is zero the behavior is undefined;
  % otherwise (a/b)*b + a%b is equal to a. If both operands are nonnegative
  % then the remainder is nonegative; if not, the sign of the remainder is
  % implementation-defined."

  % C99 requires truncation towards zero

  % proof status :-)
  % mod_impl_TCC1 : TCC Obligation

  % 5.6: C++'s % operator (i % j resp. i / j)
  mod_impl(i : int, j : {k : int | k /= 0}) : {r : int | abs(r) < abs(j)}
  div_impl(i : int, j : {k : int | k /= 0}) : int

  mod_impl_pos_range : Axiom
    Forall (i, j : nat) : j /= 0 Implies
      mod_impl(i, j) = mod(i, j)

  % proof status :-)
```

```
% div_impl_pos_range_TCC1 : TCC Obligation

div_impl_pos_range : Axiom
  Forall (i, j : nat) : j /= 0 Implies
    div_impl(i, j) = ndiv(i, j)

mod_impl_sign : Axiom
  Forall (i : int, j : int) : j /= 0 And (i < 0 Or j < 0) Implies
    mod_impl(i, j) <= 0 Or mod_impl(i, j) >= 0

div_impl_mod_impl_result : Axiom
  Forall (i : int, j : int) : j /= 0 And (i < 0 Or j < 0) Implies
    div_impl(i, j) * j + mod_impl(i, j) = i

mod(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, (range(i_typ))] =
(
  n1_ex ## lambda (n1 : (range(i_typ))):
  n2_ex ## lambda (n2 : (range(i_typ))):
  if n2 /= 0 and range(i_typ)(mod(i_typ)(mod_impl(n1, n2))) then
    ok_result(mod(i_typ)(mod_impl(n1, n2)))
  else
    fatal_result
  endif
)

% 5.6: C++'s / operator
div(i_typ)
   (n1_ex, n2_ex : ET[State, (range(i_typ))]) : ET[State, (range(i_typ))] =
(
  n1_ex ## lambda (n1 : (range(i_typ))):
  n2_ex ## lambda (n2 : (range(i_typ))):
  if n2 /= 0 And range(i_typ)(mod(i_typ)(div_impl(n1, n2))) then
    ok_result(mod(i_typ)(div_impl(n1, n2)))
  else
    fatal_result
  endif
)

% Floating point division must be handled separately.
% We use extended reals, after checking for x / 0

% To do:
% div_float(typ : Cpp_Subtype(cv(floating_point?)))
% (n1_ex, n2_ex : ET[State, (range(typ))]) : ET[State, (range(typ))] =
% (
% n1_ex ## lambda (n1 : (range(typ))):
% n2_ex ## lambda (n2 : (range(typ))):
% If n2 /= 0 And range(typ)(mod(typ)(n1 / n2)) Then
% ok_result(mod(typ)(n1 / n2))
% Else
% fatal_result
```

```
  % Endif
  % )

  % Additive operators
  %====================

  % 5.7
  plus(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
    ET[State, (range(i_typ))] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))):
    n2_ex ## lambda (n2 : (range(i_typ))):
    If range(i_typ)(mod(i_typ)(n1 + n2)) Then
      ok_result(mod(i_typ)(n1 + n2))
    Else
      fatal_result
    Endif
  );

  % To do: plus_float

  % proof status :-)
  % plus_ptr_TCC1 : TCC Obligation

  % proof status :-)
  % plus_ptr_TCC2 : TCC Obligation

  % proof status :-)
  % plus_ptr_TCC3 : TCC Obligation

  % proof status :-)
  % plus_ptr_TCC4 : TCC Obligation

  % proof status :-)
  % plus_ptr_TCC5 : TCC Obligation

  % proof status :-)
  % plus_ptr_TCC6 : TCC Obligation

  % 5.7 pt 4 ff C++'s + operator for pointers
  plus_ptr(typ : Cpp_Subtype(cv(pointer?)), i_typ)
            (p_ex : ET[State, (range_pointer(cv_base(typ)))],
             n_ex : ET[State, (range(i_typ))]) :
    ET[State, (range_pointer(cv_base(typ)))] =
  (
    p_ex ## lambda (ptr_val : (range_pointer(cv_base(typ)))) :
    n_ex ## lambda (n : (range(i_typ))) :
    If not_null?(typ)(ptr_val) And
       range_pointer(cv_base(typ))
         (add(typ)(ptr_val, n * size_of(typ(cv_base(typ))))) And
       check_bounds(typ)
         (add(typ)(ptr_val, n * size_of(typ(cv_base(typ)))))
```

```
  Then
    ok_result(add(typ)(ptr_val, n * size_of(typ(cv_base(typ)))))
  Else
    fatal_result
  Endif
)

% proof status :-)
% postinc_ptr_TCC1 : TCC Obligation

% proof status :-)
% postinc_ptr_TCC2 : TCC Obligation

% proof status :-)
% postinc_ptr_TCC3 : TCC Obligation

% proof status :-)
% postinc_ptr_TCC4 : TCC Obligation

% 5.2.6 pt 1: C++'s ++ operator for pointers to complete object
% (equivalent to p = p + 1)
postinc_ptr(pm, (typ : Cpp_Subtype(v(pointer?))))(a_ex) :
  ET[State, (range_pointer(cv_base(typ)))] =
(
  a_ex ## lambda (a : Address):
  plus_ptr(typ, uchar)(read_data(pm, dt_cv_pointer(typ))(a), ok_result(1)) ##
    lambda (ptr_val : (range_pointer(cv_base(typ)))):
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(ptr_val)
)

% 5.3.2 pt 1: C++'s ++. operator for pointers
% deferred to arithmetic expressions
preinc_ptr(pm, (typ : Cpp_Subtype(v(pointer?))))(a_ex) : ET[State, Address] =
(
  a_ex ## lambda (a : Address) :
  plus_ptr(typ, uchar)(
    read_data(pm, dt_cv_pointer(typ))(a),
    ok_result(1)) ##
      lambda (ptr_val : (range_pointer(cv_base(typ)))) :
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(a)
)

% proof status :-)
% subscript_TCC1 : TCC Obligation

% proof status :-)
% subscript_TCC2 : TCC Obligation

% proof status :-)
% subscript_TCC3 : TCC Obligation
```

```
% proof status :-)
% subscript_TCC4 : TCC Obligation

% 5.2.1: Subscripting
% "The expression E1[E2] is identical (by definition to *((E1) + (E2))
% The check_bounds function in plus and minus enforces the conditions of
% clause 5.7 pt 5
% provided the bounds-checking pointer model is implemented
subscript(pm, (typ : Cpp_Subtype(array?)), i_typ)
  (a_ex, (n_ex : ET[State, (range(i_typ))])) : ET[State, Memory_Address] =
(
  a_ex ## Lambda (a : Address) :
  deref(pm, pointer(typ(typ)))(
    plus_ptr(pointer(typ(typ)), i_typ)(
      read_data(pm, dt_cv_pointer(pointer(typ(typ))))(a),
      n_ex)
    )
)

% 5.7 C++'s - operator
minus(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
  ET[State, (range(i_typ))] =
(
  n1_ex ## lambda (n1 : (range(i_typ))):
  n2_ex ## lambda (n2 : (range(i_typ))):
  If range(i_typ)(mod(i_typ)(n1 - n2)) Then
    ok_result(mod(i_typ)(n1 - n2))
  Else
    fatal_result
  Endif
)

% To do: minus_float

% 5.7 pt 4 ff C++'s - operator for pointer - integral
minus_ptr(typ : Cpp_Subtype(cv(pointer?)), i_typ)
      (p_ex : ET[State, (range_pointer(cv_base(typ)))],
       n_ex : ET[State, (range(i_typ))]) :
  ET[State, (range_pointer(cv_base(typ)))] =
(
  p_ex ## lambda (ptr_val : (range_pointer(cv_base(typ)))) :
  n_ex ## lambda (n : (range(i_typ))) :
  If not_null?(typ)(ptr_val) And
     range_pointer(cv_base(typ))
       (add(typ)(ptr_val, -n * size_of(typ(cv_base(typ))))) And
     check_bounds(typ)
       (add(typ)(ptr_val, -n * size_of(typ(cv_base(typ)))))
  Then
    ok_result(add(typ)(ptr_val, -n * size_of(typ(cv_base(typ)))))
  Else
    fatal_result
```

```
  Endif
)

% 5.2.6 pt 2: C++'s .-- operator for pointers to complete object
postdec_ptr(pm, (typ : Cpp_Subtype(v(pointer?))))(a_ex) :
  ET[State, (range_pointer(cv_base(typ)))] =
(
  a_ex ## lambda (a : Address):
  minus_ptr(typ, uchar)(read_data(pm, dt_cv_pointer(typ))(a),
                        ok_result(1)) ##
    lambda (ptr_val : (range_pointer(cv_base(typ)))):
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(ptr_val)
)

% 5.3.2 pt 1: C++'s --. operator for pointers
predec_ptr(pm, (typ : Cpp_Subtype(v(pointer?))))(a_ex) :
    ET[State, Address] =
(
  a_ex ## lambda (a : Address) :
  minus_ptr(typ, uchar)(read_data(pm, dt_cv_pointer(typ))(a),
                        ok_result(1)) ##
    lambda (ptr_val : (range_pointer(cv_base(typ)))):
  write_data(pm, dt_cv_pointer(typ))(a, ptr_val) ##
  ok_result(a)
)

% proof status :-)
% ptr_to_nonzero_object_TCC1 : TCC Obligation

% proof status :-)
% ptr_to_nonzero_object_TCC2 : TCC Obligation

% proof status :-)
% ptr_to_nonzero_object_TCC3 : TCC Obligation

% proof status :-)
ptr_to_nonzero_object : Lemma
  Forall (typ : Cpp_Subtype(Lambda (p : (pointer?)) : Not void?(typ(p)) And
                                                      Not function?(typ(p)))) :
    (array?(typ(cv_base(typ))) Implies size_of(typ(cv_base(typ))) > 0)
  Implies
    size_of(typ(cv_base(typ))) > 0

% 5.7 pt 6 ff C++'s - operator for pointer - pointer
% "When two pointers to elements of the same array object are substracted,
% ..."
% Thus, the pointers have to have the same type.

% proof status :-)
% minus_ptr_typ_is_Cpp_Type_TCC1 : TCC Obligation
```

```
  % proof status :-)
  minus_ptr_typ_is_Cpp_Type : Lemma
    Forall (typ : Cpp_Subtype(cv(
                      Lambda (p : (pointer?)) : Not void?(typ(p)) And
                                                Not function?(typ(p))))) :
      Cpp_Type?(typ(cv_base(typ)))

  % proof status :-)
  % minus_ptr_ptr_TCC1 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC2 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC3 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC4 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC5 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC6 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC7 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC8 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC9 : TCC Obligation

  % proof status :-)
  % minus_ptr_ptr_TCC10 : TCC Obligation

  minus_ptr_ptr(typ : Cpp_Subtype(cv(
                      Lambda (p : (pointer?)) : Not void?(typ(p)) And
                                                Not function?(typ(p)))))
       (p_ex1, p_ex2 : ET[State, (range_pointer(cv_base(typ)))]) :
       ET[State, (range(int))] =
  (
    p_ex1 ## lambda (ptr_val1 : (range_pointer(cv_base(typ)))) :
    p_ex2 ## lambda (ptr_val2 : (range_pointer(cv_base(typ)))) :
    If (array?(typ(cv_base(typ))) Implies size_of(typ(cv_base(typ))) > 0) And
        % otherwise we substract pointer to incomplete arrays
        not_null?(typ)(ptr_val1) And
        not_null?(typ)(ptr_val2) And
        same_array(typ)(ptr_val1, ptr_val2) And
        up?(address_of(typ)(ptr_val1)) And
        up?(address_of(typ)(ptr_val2))
```

```
    Then
      Let difference = ndiv(offset(down(address_of(typ)(ptr_val1))) −
                            offset(down(address_of(typ)(ptr_val2))),
                            size_of(typ(cv_base(typ)))) In
        If range(int)(difference) Then
          ok_result(difference)
        Else
          fatal_result
        Endif
    Else
      fatal_result
    Endif
  )

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % Shift Operators
  %=================

  % 5.8 pt 1: "The behavior is undefined if the right operand is negative, or
  % greater than or equal to the length in bits of the promoted left operand."
  % We do not (yet) model the latter.
  %
  % We do not model that the result is implementation−defined if the first
  % operand has a signed type (for <<). The result of >> is
  % implementation−defined if the first operand has a signed type and a
  % negative value; below we require the first operand to be nonnegative.

  % 5.8 pt 1
  % type of result = type of promoted left operand (i.e., int, uint, long,
  % ulong, longlong, ulonglong)
  % undefined if E2 < 0 or E2 > length in bits (max_value_bits)


  % 5.8 pt 2
  % E1 unsigned : mod(typ(E1))(E1 << n) otherwise from_byte(to_byte(E1) << n)


  % 5.8 pt 3
  % E1 unsigned : mod(typ(E1))(E1 >> n) if E1 >= 0 ; E1 < 0 => implementation
  % defined value

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % Rational operators
  %====================

  cmp(i_typ, (R : PRED[[real, real]]))
     (n1_ex, n2_ex : ET[State, (range(i_typ))]) : ET[State, bool] =
  (
    n1_ex ## lambda (n1 : (range(i_typ))):
    n2_ex ## lambda (n2 : (range(i_typ))):
```

```
    ok_result(R(n1, n2))
  )

  % To do: cmp ops for float

  % 5.9 <
  lt(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
    ET[State, bool] =
      cmp(i_typ, <)(n1_ex, n2_ex)

  % 5.9 >
  gt(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
    ET[State, bool] =
      cmp(i_typ, >)(n1_ex, n2_ex)

  % 5.9 <=
  le(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
    ET[State, bool] =
      cmp(i_typ, <=)(n1_ex, n2_ex)

  % 5.9 >=
  ge(i_typ)(n1_ex, n2_ex : ET[State, (range(i_typ))]) :
    ET[State, bool] =
      cmp(i_typ, >=)(n1_ex, n2_ex)

  % proof status :-)
  % cmp_pointer_TCC1 : TCC Obligation

  % proof status :-)
  % cmp_pointer_TCC2 : TCC Obligation

  % Rational operators for pointers
  %-----------------------------
  % 5.9
  cmp_pointer(p_typ : Cpp_Subtype(cv(pointer?)), R : PRED[[real, real]],
              cmp_null : bool)
      (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))]) :
      ET[State, bool] =
  (
    p1_ex ## lambda (ptr_val1 : (range_pointer(cv_base(p_typ)))) :
    p2_ex ## lambda (ptr_val2 : (range_pointer(cv_base(p_typ)))) :
      If not_null?(p_typ)(ptr_val1) And
         not_null?(p_typ)(ptr_val2) Then
        If same_array(p_typ)(ptr_val1, ptr_val2) And
           up?(address_of(p_typ)(ptr_val1)) And
           up?(address_of(p_typ)(ptr_val2))
        Then
          ok_result(R(offset(down(address_of(p_typ)(ptr_val1))),
                      offset(down(address_of(p_typ)(ptr_val2)))))
        Else
          fatal_result
        Endif
```

```
      Else
        ok_result(cmp_null)
      Endif
  )

  lt_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
    (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))]) :
      ET[State, bool] =
    cmp_pointer(p_typ, <, false)(p1_ex, p2_ex)

  gt_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
    (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))]) :
      ET[State, bool] =
    cmp_pointer(p_typ, >, false)(p1_ex, p2_ex)

  le_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
    (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))]) :
      ET[State, bool] =
    cmp_pointer(p_typ, <=, true)(p1_ex, p2_ex)

  ge_ptr(p_typ : Cpp_Subtype(cv(pointer?)))
    (p1_ex, p2_ex : ET[State, (range_pointer(cv_base(p_typ)))]) :
      ET[State, bool] =
    cmp_pointer(p_typ, >=, true)(p1_ex, p2_ex)

  % Rational operators for pointers to members
  %-----------------------------------------
  % 5.9 pt 2
  % "If two pointers point to nonstatic data members of the same
  % object, or to subobjects or array elements of such members,
  % recursively, the pointer to the later declared member compares
  % greater provided the two members are not separated by an
  % access-specifier label and provided their class is not a union."

END ArithmeticExpressions

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.10: Equality operators
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

EqualityExpressions[State, Data : Type] : THEORY
BEGIN

  IMPORTING ArithmeticExpressions

  % C++'s == operator
  eq(l_ex, r_ex : ET[State, Data]) :
    ET[State, bool] =
  (
    l_ex ## lambda (l : Data) :
```

```
    r_ex ## lambda (r : Data) :
    ok_result((l = r))
  );

  % C++'s != operator
  not_equal(l_ex, r_ex : ET[State, Data]) :
    ET[State, bool] =
  (
    l_ex ## lambda (l : Data) :
    r_ex ## lambda (r : Data) :
    ok_result((l /= r))
  );

  % to do: pointers to the same function
  % to do: 5.10 pt 2 pointers to members

END EqualityExpressions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.11: Bitwise AND operator
% 5.12: Bitwise exclusive OR operator
% 5.13: Bitwise inclusive OR operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

BitwiseExpressions[State : Type] : THEORY
BEGIN

  Importing Ok_Result_Rewrite, Cpp_Types

  typ : Var Cpp_Subtype(cv(non_bool_integral_enum?))

  % 5.11. bit and &
  bitand(typ)(n1_ex, n2_ex : ET[State, (range(typ))]) :
    ET[State, (range(typ))] =
  (
    n1_ex ## lambda (n1 : (range(typ))) :
    n2_ex ## lambda (n2 : (range(typ))) :
    % Because dt(typ) is a pod for all of these types we can take any address:
    % Mem(0)
    Let a = Mem(0),
        bitand_value =
          from_byte(dt(typ))(
            apply_bitop_valuefield(value_bitmask(typ), Lambda (l : list[Byte]):
                                     binary_and(l, to_byte(dt(typ))(n2, a)))
                                   (to_byte(dt(typ))(n1, a)), a)
    In
      If up?(bitand_value) And range(typ)(down(bitand_value)) Then
        ok_result(down(bitand_value))
      Else
```

```
      fatal_result
    Endif
)

% 5.12. bit xor ^
bitxor(typ)(n1_ex, n2_ex : ET[State, (range(typ))]) :
  ET[State, (range(typ))] =
(
  n1_ex ## lambda (n1 : (range(typ))) :
  n2_ex ## lambda (n2 : (range(typ))) :
  % Because dt(typ) is a pod for all of these types we can take any address:
  % Mem(0)
  Let a = Mem(0),
      bitxor_value =
        from_byte(dt(typ))(
          apply_bitop_valuefield(value_bitmask(typ), Lambda (l : list[Byte]):
                                   binary_xor(l, to_byte(dt(typ))(n2, a)))
                                 (to_byte(dt(typ))(n1, a)), a)
  In

    If up?(bitxor_value) And range(typ)(down(bitxor_value)) Then
      ok_result(down(bitxor_value))
    Else
      fatal_result
    Endif
)

% 5.13. bit or |
bitor(typ)(n1_ex, n2_ex : ET[State, (range(typ))]) :
  ET[State, (range(typ))] =
(
  n1_ex ## lambda (n1 : (range(typ))) :
  n2_ex ## lambda (n2 : (range(typ))) :
  % Because dt(typ) is a pod for all of these types we can take any address:
  % Mem(0)
  Let a = Mem(0),
      bitor_value =
        from_byte(dt(typ))(
          apply_bitop_valuefield(value_bitmask(typ), Lambda (l : list[Byte]):
                                   binary_or(l, to_byte(dt(typ))(n2, a)))
                                 (to_byte(dt(typ))(n1, a)), a)
  In
    If up?(bitor_value) And range(typ)(down(bitor_value)) Then
      ok_result(down(bitor_value))
    Else
      fatal_result
    Endif
)

END BitwiseExpressions
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.14: Logical AND operator
% 5.15: Logical OR operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% !! left-to-right evaluation is guaranteed only for base types !!

LogicalExpressions[State : Type] : THEORY
BEGIN

  IMPORTING Ok_Result_Rewrite, Cpp_Types

  l_ex, r_ex, ex : VAR [State -> ExprResult[State, bool]]

  % 5.14
  % C++'s && operator
  and_exp(l_ex, r_ex) : [State -> ExprResult[State, bool]] =
  (
    l_ex ## lambda (l : bool):
    if l then
      r_ex ## lambda (r : bool):
      ok_result(l and r)
    else
      % 5.14.1: "Unlike &, && guarantees left-to-right evaluation: the second
      % operand is not evaluated if the first operand is false."
      ok_result(false)
    endif
  )

  % 5.15
  % C++'s || operator
  or_exp(l_ex, r_ex) : [State -> ExprResult[State, bool]] =
  (
    l_ex ## lambda (l : bool):
    if l then
      % 5.15.1: "Unlike |, || guarantees left-to-right evaluation; moreover,
      % the second operand is not evaluated if the first operand evaluates to
      % true."
      ok_result(true)
    else
      r_ex ## lambda (r : bool):
      ok_result(l or r)
    endif
  )

END LogicalExpressions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
% 5.16: Conditional operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ConditionalExpression[State, Data : Type] : THEORY
BEGIN

  IMPORTING Abstract_Read_Write_Plain, Cpp_Types

  pm : VAR Plain_Memory[State]

  d_ex1 : VAR [State -> ExprResult[State, Data]]
  d_ex2 : VAR [State -> ExprResult[State, Data]]

  b_ex : VAR [State -> ExprResult[State, bool]]

  % C++'s b_ex ? d_ex1 : d_ex2
  conditional_expr(b_ex, d_ex1, d_ex2) : [State -> ExprResult[State, Data]] =
  (
     b_ex ## lambda (b : bool) :
     If b Then
       d_ex1
     Else
       d_ex2
     Endif
  );
  % Not modelled:
  % 5.16 pt 2: one but not both expressions is void and is a throw expression
  % 5.16 pt 2 - pt 6: restrictions on conversions and restrictions on these.

End ConditionalExpression


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.17: Assignment operators
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

AssignmentExpressions[State : Type] : THEORY
BEGIN

  IMPORTING ArithmeticExpressions, BitwiseExpressions

  pm : VAR Plain_Memory[State]

  AET : Type = [State -> ExprResult[State, Address]]
  a_ex : Var AET

  p_typ : Var Cpp_Subtype(v(pointer?))
  i_typ : Var Cpp_Subtype(v(non_bool_integral_enum?))
  ci_typ : Var Cpp_Subtype(cv(non_bool_integral_enum?))
```

```
% 5.17.7: "The behavior of an expression of the form E1 op= E2 is equivalent
% to E1 = E1 op E2 except that E1 is evaluated only once."

% the result of the second expression can be of const(a_typ)

% proof status :-)
% assign_times_TCC1 : TCC Obligation

% C++'s *= operator
assign_times(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  times(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
    lambda (n : (range(i_typ))):
  write_data(pm, dt(i_typ))(a, n) ##
  ok_result(a)
)

% To do: assign_times_float

% C++'s /= operator
assign_div(pm, i_typ)
          (a_ex : AET, n_ex : ET[State, (range(i_typ))]) : ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  div(i_typ)
     (read_data(pm, dt(i_typ))(a), n_ex) ## lambda (n : (range(i_typ))) :
  write_data(pm, dt(i_typ))(a, n) ##
  ok_result(a)
)

% To do:
% assign_div_float(pm, f_typ)
% (a_ex : AET, n_ex : ET[State, (range(f_typ))]) : ET[State, Address] =
% (
% a_ex ## lambda (a : Address):
% div_float(f_typ)
% (read_data(pm, dt(f_typ))(a), n_ex) ## lambda (n : (range(f_typ))) :
% write_data(pm, dt(f_typ))(a, n) ##
% ok_result(a)
% )

% C++'s %= operator
assign_mod(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  mod(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
    lambda (n : (range(i_typ))):
  write_data(pm, dt(i_typ))(a, n) ##
```

```
  ok_result(a)
)

% C++'s += operator
assign_plus(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  plus(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
    lambda (n : (range(i_typ))):
  write_data(pm, dt(i_typ))(a, n) ##
  ok_result(a)
)

% To do: assign_plus_float

% proof status :-)
% assign_plus_ptr_TCC1 : TCC Obligation

% proof status :-)
% assign_plus_ptr_TCC2 : TCC Obligation

assign_plus_ptr(pm, p_typ, ci_typ)(a_ex : AET,
  n_ex : ET[State, (range(ci_typ))]) : ET[State, Address] =
(
  a_ex ##
    lambda (a : Address):
  plus_ptr(p_typ, ci_typ)(read_data(pm, dt_cv_pointer(p_typ))(a), n_ex) ##
    lambda (n : (range_pointer(cv_base(p_typ)))):
  write_data(pm, dt_cv_pointer(p_typ))(a, n) ##
  ok_result(a)
)

% C++'s -= operator
assign_minus(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
  ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  minus(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
    lambda (n : (range(i_typ))):
  write_data(pm, dt(i_typ))(a, n) ##
  ok_result(a)
)

% To do: assign_minus_float

assign_minus_ptr(pm, p_typ, ci_typ)(a_ex : AET,
  n_ex : ET[State, (range(ci_typ))]) : ET[State, Address] =
(
  a_ex ## lambda (a : Address):
  minus_ptr(p_typ, ci_typ)(read_data(pm, dt_cv_pointer(p_typ))(a), n_ex) ##
    lambda (n : (range_pointer(cv_base(p_typ)))) :
```

```
    write_data(pm, dt_cv_pointer(p_typ))(a, n) ##
    ok_result(a)
  )

  % C++'s &= operator
  assign_bitand(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
    ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    bitand(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
      lambda (n : (range(i_typ))) :
    write_data(pm, dt(i_typ))(a, n) ##
    ok_result(a)
  )

  % C++'s ^= operator
  assign_bitxor(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
    ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    bitxor(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
      lambda (n : (range(i_typ))):
    write_data(pm, dt(i_typ))(a, n) ##
    ok_result(a)
  )

  % C++'s |= operator
  assign_bitor(pm, i_typ)(a_ex : AET, n_ex : ET[State, (range(i_typ))]) :
    ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    bitor(i_typ)(read_data(pm, dt(i_typ))(a), n_ex) ##
      lambda (n : (range(i_typ))):
    write_data(pm, dt(i_typ))(a, n) ##
    ok_result(a)
  )

END AssignmentExpressions

AssignmentExpressions2[State, Data : Type] : Theory
Begin

  Importing AssignmentExpressions

  % Note that the assignment operators return an lvalue (i.e. an address).

  % We define assignment operators only for non class types.
  % 5.17 pt 4 "Assignment to objects of a class is defined by the copy
  % assignment operator".

  % Not modelled: 5.17 pt 8 restrictions on overlapping objects
```

```
  pm : VAR Plain_Memory[State]

  AET : Type = [State -> ExprResult[State, Address]]
  a_ex : Var AET

  % C++'s = operator
  assign(pm, (idt : (interpreted_data_type?[Data])))
        (a_ex : AET, d_ex : ET[State, Data]) : ET[State, Address] =
  (
    a_ex ## lambda (a : Address):
    d_ex ## lambda (d : Data) :
    write_data(pm, idt)(a, d) ##
    ok_result(a)
  )

END AssignmentExpressions2


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 5.18: Comma operator
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CommaExpressions[State, Data1, Data2 : Type] : THEORY
BEGIN

  IMPORTING Ok_Result_Rewrite

  l_ex : VAR [State -> ExprResult[State, Data1]]
  r_ex : VAR [State -> ExprResult[State, Data2]]

  % C++'s , operator
  comma(l_ex, r_ex) : [State -> ExprResult[State, Data2]] =
    (l_ex ## r_ex)

END CommaExpressions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Expressions: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Expressions[State : Type] : THEORY
BEGIN

  IMPORTING PrimaryExpressions,
            % PostfixExpressions,
            % UnaryExpressions,
            PointerToMemberExpressions,
```

```
        % ArithmeticExpressions,
        EqualityExpressions,
        % BitwiseExpressions,
        LogicalExpressions,
        ConditionalExpression,
        %AssignmentExpressions,
        AssignmentExpressions2,
        CommaExpressions

END Expressions
```

B.15 graph.pvs

```
Graph_Util[T : Type] : Theory
% various general stuff needed for graphs
Begin
  Importing More_List_Props, More_Function

  % proof status :-)
  every_nth : Lemma Forall(P : PRED[T], l : list[T]) :
    every(P)(l) IFF (Forall(i : below(length(l))) : P(nth(l, i)))


  % proof status :-)
  % nth_append_left_TCC1 : TCC Obligation

  % proof status :-)
  nth_append_left : Lemma
    Forall(l1, l2 : list[T], n : nat) :
      n < length(l1) Implies
        nth(append(l1, l2), n) = nth(l1, n)

  % proof status :-)
  % nth_append_right_TCC1 : TCC Obligation

  % proof status :-)
  nth_append_right : Lemma
    Forall(l1, l2 : list[T], n : nat) :
      n >= length(l1) And n < length(append(l1, l2)) Implies
        nth(append(l1, l2), n) = nth(l2, n - length(l1))


  % proof status :-)
  list_pred_card : Lemma
    Forall(l : list[T], P : PRED[T]) :
      is_finite(P) And every(P)(l) And length(l) > card(P) Implies
        Exists(n1, n2 : below(length(l))) :
          Not n1 = n2 And
          nth(l, n1) = nth(l, n2)
```

```
  % A path_list is a list with at least one element. A one
  % element path is an (pointed) empty path.
  % For the transitive closure we have of course to take path
  % with at least two elements.
  path_list?(l : list[T]) : bool = length(l) >= 1

  % proof status :-)
  % path_start_TCC1 : TCC Obligation

  path_start(l : (path_list?)) : T = car(l)

  % proof status :-)
  % path_end_TCC1 : TCC Obligation

  path_end(l : (path_list?)) : T = nth(l, length(l) -1)

  % proof status :-)
  % path_list_induction_TCC1 : TCC Obligation

  % proof status :-)
  % path_list_induction_TCC2 : TCC Obligation

  % proof status :-)
  path_list_induction: Lemma
    Forall(p: [(path_list?) -> boolean]):
      (Forall(t : T) : p(cons(t, null))) And
      (Forall(t : T, tail : (path_list?)): p(tail) Implies p(cons(t, tail)))
        Implies
          Forall (l : (path_list?)): p(l)


  concatable?(left, right : list[T]) : bool =
    path_list?(left) And path_list?(right) And
    path_end(left) = path_start(right)

  % proof status :-)
  % concat_path_TCC1 : TCC Obligation

  % proof status :-)
  % concat_path_TCC2 : TCC Obligation

  concat_path(lr : (concatable?)) : (path_list?) =
    append(Proj_1(lr), cdr(Proj_2(lr)))

  % proof status :-)
  length_concat_path : Lemma
    Forall(l, r : list[T]) : concatable?(l, r) Implies
      length(concat_path(l, r)) = length(l) + length(r) -1

End Graph_Util
```

```
Subtype_stable[T : Type] : Theory
Begin
  l : Var list[T]
  P : Var PRED[T]

  % proof status :-)
  length_stable : Lemma
    every(P)(l) Implies length[(P)](l) = length[T](l)

  % proof status :-)
  % nth_stable_TCC1 : TCC Obligation

  % proof status :-)
  nth_stable : Lemma Forall(n : below(length(l))) :
    every(P)(l) Implies
      nth[(P)](l, n) = nth[T](l,n)

End Subtype_stable


Graph_Relations[T : Type] : Theory
Begin
  Importing Graph_Util

  R : Var PRED[[T,T]]

  transitive_closure(R) : PRED[[T, T]] = Lambda(x, y : T) :
    Forall(Q : PRED[[T, T]]) : subset?(R, Q) And transitive?(Q) Implies
        Q(x,y)

  % proof status :-)
  % trans_closure_char_TCC1 : TCC Obligation

  % proof status :-)
  % trans_closure_char_TCC2 : TCC Obligation

  % proof status :-)
  % trans_closure_char_TCC3 : TCC Obligation

  % proof status :-)
  trans_closure_char : Lemma
    Forall(x, y : T) :
      transitive_closure(R)(x,y) IFF
      Exists(l : list[T]) :
        length(l) >= 2 And
        path_start(l) = x And path_end(l) = y and
        Forall(i : below(length(l) -1)) : R(nth(l, i), nth(l, i+1))

End Graph_Relations


Graph[T : Type] : Theory
```

```
Begin
  Importing Graph_Relations, Subtype_stable

  graph_type : Type = [#
    nodes : PRED[T],
    edges : PRED[[(nodes), (nodes)]]
  #]

  g : Var graph_type

  % proof status :-)
  % path?_TCC1 : TCC Obligation

  % proof status :-)
  % path?_TCC2 : TCC Obligation

  % proof status :-)
  % path?_TCC3 : TCC Obligation

  % proof status :-)
  % path?_TCC4 : TCC Obligation

  path?(g)(l : (path_list?[T])) : bool =
    (Forall(i : below(length(l))) : g`nodes(nth(l, i))) And
    (Forall(i : below(length(l) -1)) : g`edges(nth(l,i), nth(l, i+1)))

  % proof status :-)
  nodes_path_start : Lemma
    Forall(p : (path?(g))) : g`nodes(path_start(p))

  % proof status :-)
  nodes_path_end : Lemma
    Forall(p : (path?(g))) : g`nodes(path_end(p))

  % proof status :-)
  % path_head_TCC1 : TCC Obligation

  % proof status :-)
  path_head : Lemma
    Forall(p : (path?(g)), n : upto(length(p))) :
      n >= 1 Implies
        path?(g)(head(p, n))

  % proof status :-)
  % path_tail_TCC1 : TCC Obligation

  % proof status :-)
  path_tail : Lemma
    Forall(p : (path?(g)), n : upto(length(p))) :
      length(p) - n >= 1 Implies
        path?(g)(tail(p, n))
```

```
  % proof status :-)
  % path_cdr_TCC1 : TCC Obligation

  % proof status :-)
  path_cdr : Lemma
    Forall(p : (path?(g))) :
      path_list?[T](cdr(p)) Implies path?(g)(cdr(p))


  % proof status :-)
  path_concat_path : Lemma
    Forall(p1, p2 : (path?(g))) : concatable?(p1, p2) Implies
      path?(g)(concat_path(p1, p2))


  cycle_free?(g) : bool =
    Forall(x : T) : g`nodes(x) Implies
      Not transitive_closure(g`edges)(x, x)


  % proof status :-)
  trans_closure_char_graph : Lemma
    Forall(x, y : (g`nodes)) :
      transitive_closure(g`edges)(x,y) IFF
      Exists(l : (path?(g))) :
        length(l) >= 2 And
        path_start(l) = x And path_end(l) = y


  % proof status :-)
  path_cycle_free : Lemma
    cycle_free?(g) IFF
    Forall(l : (path?(g)), i1, i2 : below(length(l))) :
      nth(l, i1) = nth(l, i2) Implies i1 = i2

  % proof status :-)
  different_nodes_on_edge : Lemma
    Forall(x, y : (g`nodes)) :
      cycle_free?(g) And
      g`edges(x, y) Implies
        Not x = y

  % set of roots of a graph
  roots(g) : PRED[(g`nodes)] =
        {y : (g`nodes) | Forall(x : (g`nodes)) : Not g`edges(x,y) }


  % A tree is cycle free and each node has at most one predecessor.
  tree?(g) : bool =
    cycle_free?(g) And
    Forall(x, y, z : (g`nodes)) :
      g`edges(x,z) And g`edges(y,z) Implies x = y
```

```
% In a tree there is only one path going up from a certain node.
% So any two path ending in the same node only differ in their length.
% proof status :-)
tree_path : Lemma
  Forall(p1, p2 : (path?(g))) :
    tree?(g) And
    path_end(p1) = path_end(p2) And
    length(p1) = length(p2) Implies
      p1 = p2

% a graph is finite if its set of nodes is
finite?(g) : bool = is_finite(g`nodes)

finite_tree?(g) : bool = finite?(g) and tree?(g)

% proof status :-)
% root_path_TCC1 : TCC Obligation

root_path(g : graph_type, n : (g`nodes)) : PRED[(path?(g))] =
  Lambda(p : (path?(g))) :
      roots(g)(path_start(p)) And path_end(p) = n

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% singleton root_path
%
% In finite trees there is for every node precisely one
% path upwards to a root (root_path). This is proved in the following
% and then the function path_to_root is defined (via selection out of
% a singleton).
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% path_length_bound_TCC1 : TCC Obligation

% proof status :-)
path_length_bound : Lemma
  finite?(g) and cycle_free?(g) Implies
    Forall(p : (path?(g))) : length(p) <= card(g`nodes)

% proof status :-)
non_root_path : Lemma
  Forall(n : (g`nodes)) : roots(g)(n) OR
    Exists(p : (path?(g))) : length(p) = 2 And path_end(p) = n

% proof status :-)
% non_root_path_extend_TCC1 : TCC Obligation

% proof status :-)
non_root_path_extend : Lemma
```

```
    Forall(p : (path?(g))) : roots(g)(path_start(p)) OR
      Exists(q : (path?(g))) :
        length(q) = length(p) + 1 And
        path_end(q) = path_end(p)

  % proof status :-)
  tree_path_to_root : Lemma
    finite_tree?(g) Implies
      Forall(n : (g`nodes)) :
        Exists(p : (path?(g))) : root_path(g, n)(p)

  % proof status :-)
  tree_unique_path_to_root : Lemma
    Forall(n : (g`nodes), p1, p2 : (path?(g))) :
      tree?(g) And root_path(g, n)(p1) And root_path(g, n)(p2)
        Implies
          p1 = p2

  % proof status :-)
  root_path_singleton : Lemma Forall(n: (g`nodes)) :
    finite_tree?(g) Implies
      singleton?(root_path(g, n));

  % proof status :-)
  % path_to_root_TCC1 : TCC Obligation

  path_to_root(g : (finite_tree?), n : (g`nodes)) : (root_path(g, n)) =
        the(root_path(g, n))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % Path factoring. Every two path that start with the same node
  % have the same initial sequence.
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  % path_factor_TCC1 : TCC Obligation

  % proof status :-)
  % path_factor_TCC2 : TCC Obligation

  % proof status :-)
  % path_factor_TCC3 : TCC Obligation

  % proof status :-)
  % path_factor_TCC4 : TCC Obligation

  % proof status :-)
  path_factor : Lemma Forall (p1, p2 : (path?(g))) :
    path_start(p1) = path_start(p2) And
```

```
  (Exists(n : nat) : n < length(p1) And n < length(p2) And
      NOT nth(p1, n) = nth(p2, n))
    Implies
      Exists(q, p1t, p2t : (path?(g))) :
        concatable?(q, p1t) And concatable?(q, p2t) And
        length(p1t) >= 2 And length(p2t) >= 2 And
        p1 = concat_path(q, p1t) And
        p2 = concat_path(q, p2t) And
        nth(p1t, 0) = nth(p2t, 0) And
        NOT nth(p1t, 1) = nth(p2t, 1)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Node case distinction. Two nodes n1 and n2 can be related in the
% following way:
%
% - they are equal
% - n1 lays on the root path of n2
% - n2 lays on the root path of n1
% - the two root path lead to different roots
% (in this case the two root path are totally distinct)
% - the two root path meet at some node n3
% (in this case the two root path go to the same root)
%
% This distinction is complete and can be used to prove an arbitrary
% predicate for all pairs of nodes.
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% node_pair_distinction_TCC1 : TCC Obligation

% proof status :-)
node_pair_distinction : Lemma
  Forall(P : PRED[[(g`nodes), (g`nodes)]]) :
    finite_tree?(g) And
    (Forall(n : (g`nodes)) : P(n, n))
    And
    (Forall(n1, n2 : (g`nodes), p : (path?(g))) :
      NOT n1 = n2 And
      path_start(p) = n2 And path_end(p) = n1 And length(p) >= 2 Implies
        P(n1, n2))
    And
    (Forall(n1, n2 : (g`nodes), p : (path?(g))) :
      NOT n1 = n2 And
      path_start(p) = n1 And path_end(p) = n2 And length(p) >= 2 Implies
        P(n1, n2))
    And
    (Forall(n1, n2 : (g`nodes), p1, p2 : (path?(g))):
      NOT n1 = n2 And root_path(g, n1)(p1) And root_path(g, n2)(p2) And
      NOT path_start(p1) = path_start(p2) Implies
```

```
        P(n1, n2))
    And
    (Forall(n1, n2, n3 : (g`nodes), p1, p2 : (path?(g))):
      NOT n1 = n2 And NOT n1 = n3 And NOT n2 = n3 And
      g`edges(n3, path_start(p1)) And g`edges(n3, path_start(p2)) And
      path_end(p1) = n1 And path_end(p2) = n2 And
      NOT path_start(p1) = path_start(p2) Implies
        P(n1, n2))
    Implies
      Forall(n1, n2 : (g`nodes)) : P(n1, n2)


  % Variant for symmetric P, which leaves out the case of
  % - n2 lays on the root path of n1

  % proof status :-)
  % symmetric_node_pair_distinction_TCC1 : TCC Obligation

  % proof status :-)
  symmetric_node_pair_distinction : Lemma
    Forall(P : PRED[[(g`nodes), (g`nodes)]]) :
      finite_tree?(g) And
      symmetric?(P) And
      (Forall(n : (g`nodes)) : P(n, n))
      And
      (Forall(n1, n2 : (g`nodes), p : (path?(g))) :
        NOT n1 = n2 And
        path_start(p) = n2 And path_end(p) = n1 And length(p) >= 2 Implies
          P(n1, n2))
      And
      (Forall(n1, n2 : (g`nodes), p1, p2 : (path?(g))):
        NOT n1 = n2 And root_path(g, n1)(p1) And root_path(g, n2)(p2) And
        NOT path_start(p1) = path_start(p2) Implies
          P(n1, n2))
      And
      (Forall(n1, n2, n3 : (g`nodes), p1, p2 : (path?(g))):
        NOT n1 = n2 And NOT n1 = n3 And NOT n2 = n3 And
        g`edges(n3, path_start(p1)) And g`edges(n3, path_start(p2)) And
        path_end(p1) = n1 And path_end(p2) = n2 And
        NOT path_start(p1) = path_start(p2) Implies
          P(n1, n2))
      Implies
        Forall(n1, n2 : (g`nodes)) : P(n1, n2)

End Graph
```

B.16 hoare.pvs

```
% $Id: hoare.pvs,v 1.8.8.1.2.2 2008/05/14 14:24:58 tews Exp $
%
% Author: Tjark Weber
% (c) 2007 Radboud University
```

```
%
% Hoare logics for our shallow C++ embedding.

Hoare[State: Type] : THEORY
BEGIN

  IMPORTING State_Transformer

  P : VAR PRED[State]
  c : VAR [State -> StmtResult[State]]
  C : VAR [StmtResult[State] -> StmtResult[State]]
  Q : VAR PRED[State]
  R : Var PRED[State]

  % Currently Hoare triples with lifted statements
  % {P} C {Q}
  % are not used. They may be useful to consider partial programs on
  % intermediate states (i.e. to verify programs compositionally), but
  % otherwise, since programs are simple statements operating on states
  % (rather than on StmtResults), only Hoare triples
  % {P} c {Q}
  % are needed.

  s : VAR State

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Partial correctness: if the result is OK, it must satisfy the postcondition
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  % valid_TCC1 : TCC Obligation

  % proof status :-)
  % valid_TCC2 : TCC Obligation

  valid(P, c, Q) : bool =
    forall s: P(s) and OK?(c(s)) implies Q(state(c(s)))

  % proved - complete
  % valid_TCC1

  valid(P, C, Q) : bool =
    forall s: P(s) and OK?(C(OK(s))) implies Q(state(C(OK(s))))

  % proved - complete
  % valid_TCC2

  % proof status :-)
  valid_and : Lemma
    valid(P, c, Q) And valid(P, c, R)
```

```
  Implies
    valid(P, c, Lambda (s : State) : Q(s) And R(s))

  % proof status :-)
  valid_implied : Lemma
    valid(P, c, Q) And valid(P, c, Lambda (s : State) : Q(s) Implies R(s))
  Implies
    valid(P, c, Lambda (s : State) : Q(s) And R(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Total correctness: result must be OK and satisfy the postcondition
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Valid(P, c, Q) : bool =
    forall s: P(s) implies OK?(c(s)) and Q(state(c(s)))

  Valid(P, C, Q) : bool =
    forall s: P(s) implies OK?(C(OK(s))) and Q(state(C(OK(s))))

  % proof status :-)
  Valid_and : Lemma
    Valid(P, c, Q) And Valid(P, c, R)
  Implies
    Valid(P, c, Lambda (s : State) : Q(s) And R(s))


END Hoare
```

B.17 linear_memory.pvs

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Split range into list of aligned chunks
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% To do: move to prelude

List_Split : Theory
Begin

  Importing IA32, More_List_Props, Alignment

  % split
  %--------
  % | a a1 a2 a3 |
  % | --|-------|-------|--- |
  % =>
  % (a, --), (a1, -------), (a2, -------), (a3, ---), null
```

```
% proof status :-)
% split_TCC1 : TCC Obligation

% proof status :-)
% split_TCC2 : TCC Obligation

split(size : nat, a : Address, bl : list[Byte]) :
                          Recursive list[[Address, list[Byte]]] =
  Let e2size = expt(2, size) In
    If null?(bl) Then null Else
      If a + length(bl) <= floor(e2size)(a) + e2size Then
        cons( (a, bl), null)
      Else
        Let delta = rem(e2size)(offset(a)) In
          cons( (a, head(bl, e2size - delta)),
            split(size, floor(e2size)(a) + e2size, tail(bl, e2size - delta)))
      Endif
    Endif
   Measure length(bl)

% some correctness lemmas
% proof status :-)
split_pair_cross_size : Lemma
  Forall (size : nat, a1 : Address, bl1 : list[Byte], n : nat) :
    Let e2size = expt(2, size) In
      n = length(bl1) Implies
        every(Lambda(a2 : Address, bl2 : list[Byte]) :
                  rem(e2size)(offset(a2)) + length(bl2) <= e2size
        )(split(size, a1, bl1))

% proof status :-)
split_pair_concat : Lemma
  Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
    Let e2size = expt(2, size) In
      n = length(bl) Implies
  reduce(null,
    Lambda (e : [Address, list[Byte]], tail : list[Byte]) :
      Let head = e`2 In
        append(head, tail))(split(size, a, bl)) = bl

% proof status :-)
split_range : Lemma
  Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
    Let e2size = expt(2, size) In
      n = length(bl) Implies
    every(Lambda (a2 : Address, bl2 : list[Byte]) :
              a <= a2 And a2 + length(bl2) <= a + n)(split(size, a, bl))

% proof status :-)
split_no_null : Lemma
  Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
```

```
      Let e2size = expt(2, size) In
        n = length(bl) And cons?(bl) Implies
      every(Lambda (a2 : Address, bl2 : list[Byte]) : cons?(bl2))
           (split(size, a, bl))

  % proof status :-)
  split_null : Lemma
    Forall (size : nat, a : Address) : split(size, a, null) = null

  % proof status :-)
  % split_type_TCC1 : TCC Obligation

  % proof status :-)
  % split_type_TCC2 : TCC Obligation

  % proof status :-)
  split_type : Lemma
    Forall (size : nat, a : Address, bl : list[Byte], n : nat) :
      reg_base(min_linear)(type_of(a)) <= a And
      a + length(bl) <= reg_size(max_linear)(type_of(a))
        Implies
          every(LAMBDA (t: [Address, list[Byte]]):
                  type_of(t`1) = type_of(a) And
                  reg_base(min_linear)(type_of(a)) <= t`1 And
                  t`1 < reg_size(max_linear)(type_of(a)))
            (split(min_page, a, bl))

End List_Split

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Linear Memory
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% to do: avoid paging_data_type

Linear_Memory [Physical_memory : Type,
               (Importing Plain_Memory[Physical_memory])
               pm : Plain_Memory] : Theory
Begin

  Importing Paging_Datatype,
            PDBR_Datatype,
            Segment_Reg_Datatype,
            Address_Datatype

  Importing Abstract_Read_Write_Plain,
            List_Split

  Linear_memory : Type = Physical_memory
```

```
s : Var Linear_memory
a : Var Address
access : Var Memory_access
b : Var Byte
size : Var nat

% Protected Interface
%=====================

bits_per_level : nat = 10

pdir_lvl : nat = 0
ptab_lvl : nat = 1

% proof status :-)
% xlat_idx_TCC1 : TCC Obligation

% proof status :-)
% xlat_idx_TCC2 : TCC Obligation

% compute pe address / offset
xlat_idx(lvl : Level, base, addr : Memory_Address_4G) : Address =
  base + shift_bits_left(cut_bits(offset(addr), bus_width - (lvl + 1) *
                                    bits_per_level, bits_per_level), pe_size)

% proof status :-)
% xlat_idx_pe_aligned_TCC1 : TCC Obligation

% proof status :-)
% xlat_idx_pe_aligned_TCC2 : TCC Obligation

% proof status :-)
xlat_idx_pe_aligned : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bits_per_level + pe_size)(offset(base))
  Implies
    aligned?(pe_size)(offset(xlat_idx(lvl, base, addr)))

% proof status :-)
xlat_idx_memory_address : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bits_per_level + pe_size)(offset(base))
  Implies
    xlat_idx(lvl, base, addr) < max_linear

% proof status :-)
xlat_idx_memory_address2 : Lemma
  Forall (lvl : Level, base, addr : Memory_Address_4G) :
    aligned?(bits_per_level + pe_size)(offset(base))
  Implies
    0 <= offset(xlat_idx(lvl, base, addr))
```

```
  xlat_ofs(lvl : Level, base, addr : Memory_Address_4G) : Address =
    base + cut_bits(offset(addr), 0, bus_width − (lvl + 1) ∗ bits_per_level)

  % proof status :−)
  xlat_ofs_memory_address : Lemma
    Forall (lvl : Level, base, addr : Memory_Address_4G) :
      aligned?(bus_width − (lvl + 1) ∗ bits_per_level)(offset(base))
    Implies
      xlat_ofs(lvl, base, addr) < max_linear

  % proof status :−)
  xlat_ofs_memory_address2 : Lemma
    Forall (lvl : Level, base, addr : Memory_Address_4G) :
      aligned?(bus_width − (lvl + 1) ∗ bits_per_level)(offset(base))
    Implies
      0 <= offset(xlat_ofs(lvl, base, addr))

  % proof status :−)
  % raise_fault_TCC1 : TCC Obligation

  % raise fault
  raise_fault(priv : Memory_privilege, access : Memory_access, present : bool,
              pfa : Memory_Address_4G) :
      [Physical_memory −> ExprResult[Physical_memory, [bool, Address]]] =
    write_data(pm, address_data_type)(CR2, pfa) ##
     exception_result(Page_fault(
       (# reserved_bit_violation := false, % writing reserved bits is fatal
          privilege := priv,
          access := add(Read, access),
          present := present
        #)))

  % proof status :−)
  % translate_TCC1 : TCC Obligation

  % single level of the translation
  translate(lvl : Level, base, addr : Memory_Address_4G,
            access : Memory_access, priv : Memory_privilege)
    : [Physical_memory −> ExprResult[Physical_memory, [bool, Address]]] =
    Let pe_addr = xlat_idx(lvl, base, addr) In
      (read_data(pm, paging_data_type(lvl))(pe_addr) ##
        Lambda (pe : (range_pt(lvl))) :
          If paging_type?(lvl, pe) Then
            If present?(pe) Then
              If accessible?(pe, access) And privileged?(pe, priv) Then
                write_data(pm, paging_data_type(lvl))(pe_addr,
                  set_reference(pe, access)) ##
                    ok_result( (is_leaf?(pe), base(pe)) )
              Else
                raise_fault(priv, access, true, addr)
              Endif
```

```
          Else
            raise_fault(priv, access, false, addr)
          Endif
        Else
          fatal_result
        Endif)

  % proof status :-)
  translate_memory_address : Lemma
    Forall (lvl : Level, base, addr : Memory_Address_4G,
            access : Memory_access,
            priv : Memory_privilege, s : Linear_memory) :
      aligned?(bits_per_level + pe_size)(offset(base)) And
      OK?(translate(lvl, base, addr, access, priv)(s))
    Implies
      data(translate(lvl, base, addr, access, priv)(s))`2 < max_linear

  % proof status :-)
  translate_memory_address2 : Lemma
    Forall (lvl : Level, base, addr : Memory_Address_4G,
            access : Memory_access,
            priv : Memory_privilege, s : Linear_memory) :
      aligned?(bits_per_level + pe_size)(offset(base)) And
      OK?(translate(lvl, base, addr, access, priv)(s))
    Implies
      0 <= offset(data(translate(lvl, base, addr, access, priv)(s))`2)


  % proof status :-)
  % translate_result_leaf_TCC1 : TCC Obligation

  % proof status :-)
  % translate_result_leaf_TCC2 : TCC Obligation

% !!! translate_result_leaf_TCC1 : original long running proof is nondefault !!
  % proof status :-)
  translate_result_leaf : Lemma
    Forall (lvl : Level, base, addr : Memory_Address_4G,
            access : Memory_access,
            priv : Memory_privilege, s : Linear_memory) :
      aligned?(bits_per_level + pe_size)(offset(base)) And
      OK?(translate(lvl, base, addr, access, priv)(s)) And
      data(translate(lvl, base, addr, access, priv)(s))`1
    Implies
      aligned?(bus_width - bits_per_level * (lvl + 1))
              (offset(data(translate(lvl, base, addr, access, priv)(s))`2))

  % proof status :-)
  % translate_result_no_leaf_TCC1 : TCC Obligation

  % proof status :-)
  translate_result_no_leaf : Lemma
```

```
    Forall (lvl : Level, base, addr : Memory_Address_4G,
            access : Memory_access,
            priv : Memory_privilege, s : Linear_memory) :
      aligned?(bits_per_level + pe_size)(offset(base)) And
      OK?(translate(lvl, base, addr, access, priv)(s)) And
      Not data(translate(lvl, base, addr, access, priv)(s))`1
    Implies
      aligned?(bits_per_level + pe_size)
              (offset(data(translate(lvl, base, addr, access, priv)(s))`2))


  % proof status :-)
  % linear_resolve_TCC1 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC2 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC3 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC4 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC5 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC6 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC7 : TCC Obligation

  % proof status :-)
  % linear_resolve_TCC8 : TCC Obligation

  % translate
  linear_resolve(addr : Memory_Address_4G, access : Memory_access)
              : [Linear_memory -> ExprResult[Linear_memory, Address]] =
    % 1. get current privilege level from cs
    (read_data(pm, segment_reg_data_type)(CS) ##
      Lambda (cs : Segment_Reg_type) :
        Let priv = segment_to_priv(cs) In
      % 2. read PDBR
      (read_data(pm, pdbr_data_type)(PDBR) ##
        Lambda (pdbr : Pdbr_type) :
        % 3. PDE lookup
        Let lvl = pdir_lvl In
          (translate(lvl, pdbr`base_addr, addr, access, priv) ##
            Lambda (leaf : bool,
                    base : {b : Memory_Address_4G |
              If leaf Then
                aligned?(bus_width - bits_per_level * (lvl + 1))(offset(b))
```

```
            Else
              aligned?(bits_per_level + pe_size)(offset(b))
            Endif}) :
          If leaf Then
            ok_result(xlat_ofs(lvl, base, addr))
          Else
            Let lvl = ptab_lvl In
              (translate(lvl, base, addr, access, priv) ##
                Lambda (leaf : bool,
                        base : Memory_Address_4G) :
                  ok_result(xlat_ofs(lvl, base, addr)))
            Endif)))

% proof status :-)
linear_resolve_memory_address : Judgement
  linear_resolve(addr : Memory_Address_4G, access : Memory_access) Has_Type
    [Linear_memory -> ExprResult[Linear_memory, Memory_Address_4G]]

% side effect helpers
linear_read_side_effect_in_page(a : Memory_Address_4G, bl : list[Byte])(s) :
                                        ExprResult[Linear_memory, list[Byte]] =
  (linear_resolve(a, Read) ##
    Lambda(pa : Memory_Address_4G)(ns : Linear_memory) :
      memory_read_side_effect(pm`mem)(pa, bl, true)(s))(s)

linear_write_side_effect_in_page(a : Memory_Address_4G, bl : list[Byte])(s) :
                                        ExprResult[Linear_memory, list[Byte]] =
  (linear_resolve(a, Write) ##
    Lambda(pa : Memory_Address_4G)(ns : Linear_memory) :
      memory_write_side_effect(pm`mem)(pa, bl, true)(s))(s)

apply_side_effects(l : list[[Memory_Address_4G, list[Byte]]],
                   f : [Memory_Address_4G, list[Byte] ->
                [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]])(s) :
                                          ExprResult[Linear_memory, list[Byte]] =
  reduce(ok_result(null),
    Lambda (e : [Memory_Address_4G, list[Byte]],
            r : [Linear_memory -> ExprResult[Linear_memory, list[Byte]]]) :
      (r ##
        Lambda (tail : list[Byte]) :
          (f(e) ##
            Lambda (head : list[Byte]) : ok_result(append(head, tail))))
    )(l)(s)
  % Note that the element with the highest address is evaluated first

% Public Interface
%=================

% proof status :-)
% linear_read_TCC1 : TCC Obligation

% proof status :-)
```

```
  % linear_read_TCC2 : TCC Obligation

  % proof status :-)
  % linear_read_TCC3 : TCC Obligation

  % read & write
  linear_read(a) : [Linear_memory -> ExprResult[Linear_memory, Byte]] =
    If in_memory(min_linear, max_linear)(a) Then
      If Mem?(type_of(a)) Then
        linear_resolve(a, Read) ##
          Lambda (pa : Address) :
            memory_read(pm`mem)(pa)
      Else
        memory_read(pm`mem)(a)
      Endif
    Else
      fatal_result
    Endif

  linear_write(a, b) : [Linear_memory -> ExprResult[Linear_memory, Unit]] =
    If in_memory(min_linear, max_linear)(a) Then
      If Mem?(type_of(a)) Then
        linear_resolve(a, Write) ##
          Lambda (pa : Address) :
            memory_write(pm`mem)(pa, b)
      Else
        memory_write(pm`mem)(a, b)
      Endif
    Else
      fatal_result
    Endif

  % proof status :-)
  % linear_read_side_effect_TCC1 : TCC Obligation

  % Linear Side Effects
  linear_read_side_effect(a : Address, bl : list[Byte], cp : bool)(s)
                                        : ExprResult[Linear_memory, list[Byte]] =
    If null?(bl) Or
       (reg_base(min_linear)(type_of(a)) <= a And
        a + length(bl) <= reg_size(max_linear)(type_of(a))) Then
      If Mem?(type_of(a)) Then
        % split list in page contained lists
        apply_side_effects(split(min_page, a, bl),
                           linear_read_side_effect_in_page)(s)
      Else
        memory_read_side_effect(pm`mem)(a, bl, cp)(s)
      Endif
    Else
      Fatal
    Endif
```

```
  linear_write_side_effect(a : Address, bl : list[Byte], cp : bool)(s)
                                        : ExprResult[Linear_memory, list[Byte]] =
    If null?(bl) Or
       (reg_base(min_linear)(type_of(a)) <= a And
        a + length(bl) <= reg_size(max_linear)(type_of(a))) Then
      If Mem?(type_of(a)) Then
        % split list in page contained lists
        apply_side_effects(split(min_page, a, bl),
                           linear_write_side_effect_in_page)(s)
      Else
        memory_write_side_effect(pm`mem)(a, bl, cp)(s)
      Endif
    Else
      Fatal
    Endif

  % The memory struct
  linear_pm : Memory_struct[Linear_memory] = (#
     memory_read := linear_read,
     memory_write := linear_write,

     memory_read_side_effect := linear_read_side_effect,
     memory_write_side_effect := linear_write_side_effect
  #)

END Linear_Memory
```

B.18 memory.pvs

```
% $Id: memory.pvs,v 1.55.6.1.4.3 2008/05/14 14:24:59 tews Exp $

Address_Util : Theory
Begin
  IMPORTING IA32
  IMPORTING More_Sets_Lemmas % for disjoint_mono
  Importing More_List_Props

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % address blocks
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % the set of addresses addr <= a < addr + size
  address_block(addr : Address, size : nat) : PRED[Address] =
    Lambda(a : Address) :
        type_of(addr) = type_of(a) And
        offset(addr) <= offset(a) AND
        offset(a) < offset(addr) + size

  % a lemma needed in the induction below
```

```
  % proof status :-)
  address_block_subset_1 : Lemma Forall(addr : Address, size : nat) :
    subset?(address_block(addr + 1, size),
            address_block(addr, 1 + size))

  % proof status :-)
  address_block_subset_2 : Lemma
    Forall(addresses : PRED[Address], addr : Address, size : nat) :
      subset?(address_block(addr, size + 1), addresses) Implies
        subset?(address_block(addr + 1, size), addresses)

  blocks_disjoint?(addr1 : Address, size1 : nat,
                   addr2 : Address, size2 : nat) : bool =
    type_of(addr1) /= type_of(addr2) Or
      addr1 + size1 <= addr2 OR
        addr2 + size2 <= addr1

  % proof status :-)
  blocks_disjoint_disjoint : Lemma
    Forall (addr1, addr2 : Address, size1, size2 : nat) :
      size1 > 0 And size2 > 0 Implies
        blocks_disjoint?(addr1, size1, addr2, size2) =
          disjoint?(address_block(addr1, size1), address_block(addr2, size2))

  % proof status :-)
  blocks_disjoint_symmetric : Lemma
    Forall (addr1, addr2 : Address, size1, size2 : nat) :
      blocks_disjoint?(addr1, size1, addr2, size2) =
        blocks_disjoint?(addr2, size2, addr1, size1)


  % proof status :-)
  blocks_in_larger_set : Lemma
    Forall (addr1, addr2 : Address, size1, size2 : nat,
                                                addresses : PRED[Address]) :
      subset?(address_block(addr1, size1), addresses) And
      subset?(address_block(addr2, size2), addresses) And
      blocks_disjoint?(addr1, size1, addr2, size2)
        Implies
          subset?(address_block(addr2, size2),
                  difference(addresses, address_block(addr1, size1)))


  blocks_pairwise_disjoint(blocks : list[[Address, posnat]]) : bool =
    Forall(addr_1, addr_2 : Address, size_1, size_2 : posnat) :
      member((addr_1, size_1), blocks) And
      member((addr_2, size_2), list_remove((addr_1, size_1), blocks))
        Implies
          blocks_disjoint?(addr_1, size_1, addr_2, size_2)

  % proof status :-)
  blocks_pairwise_disjoint_add : Lemma
```

```
    Forall(addr : Address, size : posnat, blocks : list[[Address, posnat]]) :
      blocks_pairwise_disjoint(blocks) And
      (Forall(oaddr : Address, osize : posnat) :
        member((oaddr, osize), blocks) Implies
              blocks_disjoint?(addr, size, oaddr, osize))
      Implies
        blocks_pairwise_disjoint(cons((addr, size), blocks))

  % proof status :-)
  blocks_pairwise_disjoint_remove : Lemma
    Forall(addr : Address, size : posnat, blocks : list[[Address, posnat]]) :
      blocks_pairwise_disjoint(blocks) Implies
        blocks_pairwise_disjoint(list_remove((addr, size), blocks))

  block_is_free(block : PRED[Address],
                alloc_points : list[[Address, posnat]]) : bool =
    Forall(addr : Address, size : posnat) :
      member((addr, size), alloc_points) Implies
        disjoint?(block, address_block(addr, size))

  % proof status :-)
  subset_block_is_free : Lemma
    Forall(b1, b2 : PRED[Address], alloc_points : list[[Address, posnat]]) :
      subset?(b1, b2) And
      block_is_free(b2, alloc_points) Implies
        block_is_free(b1, alloc_points)

  % proof status :-)
  block_is_free_add : Lemma
    Forall(block : PRED[Address], alloc_points : list[[Address, posnat]],
                                                  addr : Address, size : posnat) :
      block_is_free(block, alloc_points) And
      disjoint?(block, address_block(addr, size))
        Implies
          block_is_free(block, cons((addr, size), alloc_points))

End Address_Util



Memory[State : Type] : Theory
%
% memory is a common memory abstraction
% it comes with read and write functions
%
% this theory defines the notion of memory and
% some utility functions/lemmas
%
Begin
  IMPORTING Unit, Address_Util, State_Transformer

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%
% memory definition
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Memory_struct : Type =
  [#
    memory_read : [Address ->
                                  [State -> ExprResult[State, Byte]]],
    memory_write : [Address, Byte ->
                                  [State -> ExprResult[State, Unit]]],

    memory_read_side_effect : [Address, list[Byte], bool ->
                            [State -> ExprResult[State, list[Byte]]]],
    memory_write_side_effect : [Address, list[Byte], bool ->
                            [State -> ExprResult[State, list[Byte]]]]
  #]

s : Var State
pm : Var Memory_struct
bl : Var list[Byte]
addr : Var Address
size : Var nat


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% write_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% memory_write_list_nse_TCC1 : TCC Obligation

% proof status :-)
% memory_write_list_nse_TCC2 : TCC Obligation

% side effect free list write
memory_write_list_nse(pm)(addr, bl) : Recursive
                                  [State -> ExprResult[State, Unit]] =
  Cases bl of
    null : ok_result(unit),
    cons(b, rl) :
      (memory_write(pm)(addr, b) ##
         memory_write_list_nse(pm)(addr + 1, cdr(bl)))
  EndCases
 Measure length(bl)

% apply write side effect before register is modified
memory_write_list(pm)(addr, bl) : [State -> ExprResult[State, Unit]] =
  memory_write_side_effect(pm)(addr, bl, false) ##
    Lambda (bl1 : list[Byte]) :
```

```
      memory_write_list_nse(pm)(addr, bl1)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% read_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% memory_read_list_nse_TCC1 : TCC Obligation

% proof status :-)
% memory_read_list_nse_TCC2 : TCC Obligation

% side effect free list read
memory_read_list_nse(pm)(addr, size) : Recursive
                                       [State -> ExprResult[State, list[Byte]]] =
  If size > 0 Then
    memory_read(pm)(addr) ##
      Lambda (b : Byte) :
        (memory_read_list_nse(pm)(addr + 1, size - 1) ##
           Lambda (res_list: list[Byte]) :
            ok_result(cons(b, res_list)))

  Else
    ok_result(null)
  Endif
 Measure size

% apply side effect on read content
memory_read_list(pm)(addr, size) : [State -> ExprResult[State, list[Byte]]] =
  memory_read_list_nse(pm)(addr, size) ##
    Lambda (bl1 : list[Byte]) :
      memory_read_side_effect(pm)(addr, bl1, false)

% proof status :-)
memory_read_list_ok_length : Lemma
  OK?(memory_read_list_nse(pm)(addr, size)(s))
  IMPLIES
    length(data(memory_read_list_nse(pm)(addr, size)(s))) = size


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read transformers (sets of memory_read's)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

addresses : Var PRED[Address]

% the set of state transformers that read at addresses
```

```
memory_read_transformers(pm, addresses) :
                              PRED[[State -> SuperResult[State]]] =
  Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address) : addresses(a) AND
      q = expr_2_super(memory_read(pm)(a))


% the obvious rewrite lemma
% proof status :-)
memory_read_transformers_memory_read : Lemma Forall(addr : Address) :
  addresses(addr)
  IMPLIES
    memory_read_transformers(pm, addresses)
                          (expr_2_super(memory_read(pm)(addr)))


% Monotonicity
% proof status :-)
memory_read_transformers_mono : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) IMPLIES
      subset?(memory_read_transformers(pm, addresses_1),
        memory_read_transformers(pm, addresses_2))

% proof status :-)
memory_read_transformers_union : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]):
    union(memory_read_transformers(pm, addresses_1),
          memory_read_transformers(pm, addresses_2))
      =
        memory_read_transformers(pm, union(addresses_1, addresses_2))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_write transformers (sets of memory_write's)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% the set of state transformers that read at addresses
memory_write_transformers(pm, addresses) :
                               PRED[[State -> SuperResult[State]]] =
  Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address, b : Byte) :
              addresses(a) AND q = expr_2_super(memory_write(pm)(a,b))


% the obvious rewrite lemma
% proof status :-)
memory_write_transformers_memory_write : Lemma
  Forall(addr : Address, b : Byte) :
  addresses(addr)
```

```
  IMPLIES
    memory_write_transformers(pm, addresses)
                        (expr_2_super(memory_write(pm)(addr, b)))


% Monotonicity
% proof status :-)
memory_write_transformers_mono : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) IMPLIES
      subset?(memory_write_transformers(pm, addresses_1),
        memory_write_transformers(pm, addresses_2))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% side effect transformers
% (sets of memory_read_side_effect and memory_write_side_effect
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

states : Var PRED[State]
transformers : Var PRED[[State -> ExprResult[State, list[Byte]]]]

memory_read_side_effect_super_transformers(pm, addresses) :
                                        PRED[[State -> SuperResult[State]]] =
  Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address, bl : list[Byte], cp : bool) :
      subset?(address_block(a, length(bl)), addresses) And
      q = expr_2_super(memory_read_side_effect(pm)(a, bl, cp))

% proof status :-)
memory_read_side_effect_super_transformers_memory_read_side_effect : Lemma
  Forall(a : Address, bl : list[Byte], cp : bool) :
    subset?(address_block(a, length(bl)), addresses) Implies
      memory_read_side_effect_super_transformers(pm, addresses)
        (expr_2_super(memory_read_side_effect(pm)(a, bl, cp)))

% proof status :-)
memory_read_side_effect_super_transformers_mono : Lemma
  Forall(addresses_1, addresses_2 : PRED[Address]) :
    subset?(addresses_1, addresses_2) IMPLIES
      subset?(memory_read_side_effect_super_transformers(pm, addresses_1),
              memory_read_side_effect_super_transformers(pm, addresses_2))

memory_write_side_effect_super_transformers(pm, addresses) :
                                        PRED[[State -> SuperResult[State]]] =
  Lambda(q : [State -> SuperResult[State]]) :
    Exists(a : Address, bl : list[Byte], cp : bool) :
      subset?(address_block(a, length(bl)), addresses) And
      q = expr_2_super(memory_write_side_effect(pm)(a, bl, cp))
```

```
  % proof status :-)
memory_write_side_effect_super_transformers_memory_write_side_effect : Lemma
    Forall(a : Address, bl : list[Byte], cp : bool) :
      subset?(address_block(a, length(bl)), addresses) Implies
        memory_write_side_effect_super_transformers(pm, addresses)
          (expr_2_super(memory_write_side_effect(pm)(a, bl, cp)))

  % proof status :-)
memory_write_side_effect_super_transformers_mono : Lemma
    Forall(addresses_1, addresses_2 : PRED[Address]) :
      subset?(addresses_1, addresses_2) IMPLIES
        subset?(memory_write_side_effect_super_transformers(pm, addresses_1),
                memory_write_side_effect_super_transformers(pm, addresses_2))


se_transformer : Var
        [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

se_transformer1 : Var
        [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

se_transformer2 : Var
        [Address, list[Byte], bool -> [State -> ExprResult[State, list[Byte]]]]

se_super_transformers(addresses : PRED[Address],
                      se_transformer : [Address, list[Byte], bool ->
                                        [State -> ExprResult[State, list[Byte]]]]) :
                                          PRED[[State -> SuperResult[State]]] =
    Lambda (q : [State -> SuperResult[State]]) :
      Exists(a : Address, bl : list[Byte], cp : bool) :
        subset?(address_block(a, length(bl)), addresses) And
        q = expr_2_super(se_transformer(a, bl, cp))

  % side effect transformers do not change the data read or to be written
side_effect_content_unchanged(addresses : PRED[Address], states,
                              se_transformer) : bool =
    Forall(s : State, a : Address, bl : list[Byte], cp : bool) :
      states(s) And
      subset?(address_block(a, length(bl)), addresses) And
      OK?(se_transformer(a, bl, cp)(s)) Implies
        data(se_transformer(a, bl, cp)(s)) = bl

  % simple access lemma
  % proof status :-)
side_effect_content_unchanged_content : Lemma
    Forall(addresses : PRED[Address],
           s : State, a : Address, bl : list[Byte], cp : bool) :
      side_effect_content_unchanged(addresses, states, se_transformer) And
      states(s) And
      subset?(address_block(a, length(bl)), addresses) And
      OK?(se_transformer(a, bl, cp)(s)) Implies
        data(se_transformer(a, bl, cp)(s)) = bl
```

```
  % proof status :-)
  side_effect_content_unchanged_mono : Lemma
    Forall (addresses_1, addresses_2 : PRED[Address]) :
      subset?(addresses_1, addresses_2) And
      side_effect_content_unchanged(addresses_2, states, se_transformer)
        Implies
      side_effect_content_unchanged(addresses_1, states, se_transformer)

  % proof status :-)
  side_effect_content_unchanged_composition : Lemma
    transformer_invariant?(states,
                           se_super_transformers(addresses, se_transformer1)) And
    side_effect_content_unchanged(addresses, states, se_transformer1) And
    side_effect_content_unchanged(addresses, states, se_transformer2)
  Implies
    side_effect_content_unchanged(addresses, states,
      Lambda(a : Address, bl : list[Byte], cp : bool) :
        se_transformer1(a, bl, cp) ## Lambda (bl1 : list[Byte]) :
        se_transformer2(a, bl1, cp))

end Memory


Memory_Change[State : Type] : Theory
%
% this theory defines the unchanged abstraction
%
Begin
  IMPORTING Memory

  pm : Var Memory_struct[State]
  states : VAR PRED[State]
  transformers : Var PRED[[State -> SuperResult[State]]]
  addresses : Var PRED[Address]

  % proof status :-)
  % unchanged_memory_invariant?_TCC1 : TCC Obligation

  % Unchanged abstraction: For
  % - an arbitrary memory,
  % - a set of states
  % - a set of state transformers and
  % - a set of addresses
  % unchanged_memory_invariant? holds if
  % - states form an invariant wrt. all elements of transformers
  % - all transformers leave the values at all addresses unchanged
  unchanged_memory_invariant?(pm, states, transformers, addresses) : bool =
    transformer_invariant?(states, transformers) AND
    Forall(s : State, q : [State -> SuperResult[State]], a : Address) :
      states(s) AND transformers(q) AND addresses(a) AND
      OK?(q(s)) AND
```

```
    OK?(memory_read(pm)(a)(s)) AND
    OK?(memory_read(pm)(a)(state(q(s))))
    IMPLIES
      data(memory_read(pm)(a)(state(q(s)))) =
        data(memory_read(pm)(a)(s))


% proof status :-)
% unchanged_memory_invariant_unchanged_TCC1 : TCC Obligation

% access lemma
% proof status :-)
unchanged_memory_invariant_unchanged : Lemma
  Forall(s : State, q : [State -> SuperResult[State]], a : Address) :
    unchanged_memory_invariant?(pm, states, transformers, addresses) AND
    states(s) AND transformers(q) AND addresses(a) AND
    OK?(q(s)) AND
    OK?(memory_read(pm)(a)(s)) AND
    OK?(memory_read(pm)(a)(state(q(s))))
    IMPLIES
      data(memory_read(pm)(a)(state(q(s)))) =
        data(memory_read(pm)(a)(s))

% proof status :-)
unchanged_memory_invariant_invariant : Lemma
  unchanged_memory_invariant?(pm, states, transformers, addresses) Implies
    transformer_invariant?(states, transformers)

% Monotonicity lemma
% proof status :-)
unchanged_memory_invariant_mono : Lemma
  Forall(transformers_1, transformers_2 : PRED[[State -> SuperResult[State]]],
         addresses_1, addresses_2 : PRED[Address]) :
    subset?(transformers_1, transformers_2) AND
    subset?(addresses_1, addresses_2) AND
    unchanged_memory_invariant?(pm, states, transformers_2, addresses_2)
IMPLIES
  unchanged_memory_invariant?(pm, states, transformers_1, addresses_1)


% proof status :-)
% unchanged_memory_invariant_next_ok_TCC1 : TCC Obligation

% invariant lemma
% proof status :-)
unchanged_memory_invariant_next_ok: LEMMA
  FORALL (s: State, q: [State -> SuperResult[State]]):
    unchanged_memory_invariant?(pm, states, transformers, addresses) AND
     states(s) AND transformers(q) AND has_next_state(q(s))
     IMPLIES states(state(q(s)))

% lemma for union of transformers
```

```
  % proof status :-)
  unchanged_memory_invariant_union_transformers : Lemma
    Forall(transformers_1, transformers_2 :
             PRED[[State -> SuperResult[State]]]) :
      unchanged_memory_invariant?(pm, states, transformers_1, addresses) AND
      unchanged_memory_invariant?(pm, states, transformers_2, addresses)
    IMPLIES
      unchanged_memory_invariant?(pm, states,
                                  union(transformers_1, transformers_2), addresses)

  % lemma for union of address sets
  % proof status :-)
  unchanged_memory_invariant_union_addresses : Lemma
    Forall(addresses_1, addresses_2 : PRED[Address]) :
      unchanged_memory_invariant?(pm, states, transformers, addresses_1) AND
      unchanged_memory_invariant?(pm, states, transformers, addresses_2)
    IMPLIES
      unchanged_memory_invariant?(pm, states, transformers,
                                            union(addresses_1, addresses_2))

  % An empty address set does not automatically imply an
  % unchanged_memory_invariant, because states and transformers must
  % form a transformer invariant (independent of any address).
  % proof status :-)
  unchanged_memory_invariant_empty : Lemma
    empty?(states) OR empty?(transformers) Implies
      unchanged_memory_invariant?(pm, states, transformers, addresses)

  % proof status :-)
  unchanged_memory_invariant_all_transformers : Lemma
    (Forall (q : [State -> SuperResult[State]]) : transformers(q) Implies
       unchanged_memory_invariant?(pm, states, singleton(q), addresses))
    Implies
      unchanged_memory_invariant?(pm, states, transformers, addresses)

End Memory_Change



Memory_Change_3[State : Type, Data : Type] : Theory
% repeat a few lemmas from above for non-super transformers
Begin
  Importing Memory_Change

  pm : Var Memory_struct[State]
  states : VAR PRED[State]
  transformers : Var PRED[[State -> SuperResult[State]]]
  addresses : Var PRED[Address]

  % proof status :-)
  % expr_unchanged_memory_invariant_unchanged_TCC1 : TCC Obligation
```

```
  % proof status :-)
  expr_unchanged_memory_invariant_unchanged : Lemma
    Forall(s : State, q : [State -> ExprResult[State, Data]], a : Address) :
      unchanged_memory_invariant?(pm, states, transformers, addresses) AND
      states(s) AND transformers(expr_2_super(q)) AND addresses(a) AND
      OK?(q(s)) AND
      OK?(memory_read(pm)(a)(s)) AND
      OK?(memory_read(pm)(a)(state(q(s))))
      IMPLIES
        data(memory_read(pm)(a)(state(q(s)))) =
          data(memory_read(pm)(a)(s))

  % proof status :-)
  % expr_unchanged_memory_invariant_next_ok_TCC1 : TCC Obligation

  % proof status :-)
  expr_unchanged_memory_invariant_next_ok: LEMMA
    FORALL (s: State, q: [State -> ExprResult[State, Data]]):
      unchanged_memory_invariant?(pm, states, transformers, addresses) AND
       states(s) AND transformers(expr_2_super(q)) AND has_next_state(q(s))
       IMPLIES states(state(q(s)))


End Memory_Change_3

Memory_Change_4[State : Type, Data1, Data2 : Type] : Theory
Begin

  Importing Memory_Change_3

  pm : Var Memory_struct[State]
  states : Var PRED[State]
  transformers_1 : Var PRED[[State -> SuperResult[State]]]
  transformers_2 : Var PRED[[State -> SuperResult[State]]]
  addresses : Var PRED[Address]

  % proof status :-)
  expr_unchanged_memory_invariant_composition : Lemma
    Forall (q1 : [State -> ExprResult[State, Data1]],
            q2 : [State -> ExprResult[State, Data2]]) :
      transformers_1(expr_2_super(q1)) And transformers_2(expr_2_super(q2)) And
      transformers_ok?(states, memory_read_transformers(pm, addresses)) And
      unchanged_memory_invariant?(pm, states, transformers_1, addresses) And
      unchanged_memory_invariant?(pm, states, transformers_2, addresses)
    Implies
      unchanged_memory_invariant?(pm, states, singleton(expr_2_super(q1 ## q2)), addresses)

  % proof status :-)
  fexpr_unchanged_memory_invariant_composition : Lemma
    Forall (q1 : [State -> ExprResult[State, Data1]],
            q2 : [Data1 -> [State -> ExprResult[State, Data2]]],
            P : PRED[Data1]) :
```

```
      transformers_1(expr_2_super(q1)) And
      transformers_ok?(states, memory_read_transformers(pm, addresses)) And
      unchanged_memory_invariant?(pm, states, transformers_1, addresses) And
      (Forall (s : (states)) : OK?(q1(s)) Implies P(data(q1(s)))) And
      (Forall (d : (P)) : unchanged_memory_invariant?(pm, states,
                                                      singleton(expr_2_super(q2(d))), addresses))
    Implies
      unchanged_memory_invariant?(pm, states, singleton(expr_2_super(q1 ## q2)), addresses)

End Memory_Change_4

Memory_Change_2[State : Type] : Theory
%
% Here the unchanged abstraction is further specialized.
% There is also a changed abstraction (for memory writes)
% (all this cannot go in the previous theory because of
% the type parameter Data there)
%
% in the end all this is used to prove results about
% memory_read_list after memory_write_list
% and
% memory_read_list after memory_read_list
%
Begin

  IMPORTING Memory_Change_3

  pm : Var Memory_struct[State]

  states : VAR PRED[State]
  addresses : Var PRED[Address]


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % special unchanged for memory_write
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % The following is a variation of
  %
  % unchanged_memory_invariant?(pm, states,
  % memory_write_transformers(pm, addresses),
  % addresses)
  %
  % The variation is that it excludes the obviously wrong cases:
  % writing at a _changes_ address a.
  unchanged_memory_write_invariant?(pm, states, addresses) : bool =
    Forall(waddr : Address, b : Byte) :
      addresses(waddr) IMPLIES
        unchanged_memory_invariant?(pm, states,
                singleton(expr_2_super(memory_write(pm)(waddr,b))),
```

```
                remove(waddr, addresses))

  % Monotonicity
  % proof status :-)
  unchanged_memory_write_invariant_mono_addresses : Lemma
    Forall(addresses_1, addresses_2 : PRED[Address]) :
      subset?(addresses_1, addresses_2) AND
      unchanged_memory_write_invariant?(pm, states, addresses_2)
      IMPLIES
        unchanged_memory_write_invariant?(pm, states, addresses_1)

  % connect unchanged_memory_write_invariant? with transformer_invariant?
  % proof status :-)
  unchanged_memory_write_invariant_transformer_invariant : Lemma
    unchanged_memory_write_invariant?(pm, states, addresses)
      IMPLIES
        transformer_invariant?(states, memory_write_transformers(pm,addresses))


  % connect unchanged_memory_write_invariant with unchanged_memory_invariant?
  % for a subset of unchanged addresses
  % proof status :-)
  unchanged_memory_write_invariant_unchanged_memory_invariant : Lemma
    Forall(all_addresses, unchanged_addresses,
           changed_addresses : PRED[Address]) :
      unchanged_memory_write_invariant?(pm, states, all_addresses) AND
      subset?(unchanged_addresses, all_addresses) AND
      subset?(changed_addresses, all_addresses) AND
      disjoint?(unchanged_addresses, changed_addresses)
    IMPLIES
      unchanged_memory_invariant?(pm, states,
                                  memory_write_transformers(pm, changed_addresses),
                                        unchanged_addresses)

  % connect unchanged_memory_invariant and unchanged_memory_write_invariant
  % to a larger unchanged_memory_invariant
  % proof status :-)
  unchanged_memory_invariant_union : Lemma
    Forall (ro_addr, rw_addr, mod_addr : PRED[Address]) :
      unchanged_memory_invariant?(pm, states,
        memory_write_transformers(pm, rw_addr), ro_addr) And
      unchanged_memory_write_invariant?(pm, states, rw_addr) And
      subset?(mod_addr, rw_addr)
        Implies
          unchanged_memory_invariant?(pm, states,
                        memory_write_transformers(pm, mod_addr),
                        difference(union(ro_addr, rw_addr), mod_addr))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % some invariant lemmas for write_list
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% wrt transformer_invariant
% proof status :-)
transformer_invariant_write_list_nse : Lemma
  Forall(waddr : Address, bl : list[Byte], s : State) :
    transformer_invariant?(states,
              memory_write_transformers(pm, address_block(waddr, length(bl))))
    AND
    states(s)
  IMPLIES
    result_pred(states)
              (expr_2_super(memory_write_list_nse(pm)(waddr, bl))(s))

% proof status :-)
transformer_invariant_write_list : Lemma
  Forall(waddr : Address, bl : list[Byte], s : State) :
    transformer_invariant?(states,
      add(
        expr_2_super(memory_write_side_effect(pm)(waddr, bl, false)),
        memory_write_transformers(pm, address_block(waddr, length(bl)))))
    And
    side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                  memory_write_side_effect(pm))
    And
    states(s)
  Implies
    result_pred(states)(expr_2_super(memory_write_list(pm)(waddr, bl))(s))

% wrt unchanged_memory_invariant? and read_memory
% see below


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% changed abstraction for memory_read/memory_write
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% changed_memory_invariant?_TCC1 : TCC Obligation

changed_memory_invariant?(pm, states, addresses) : bool =
  transformer_invariant?(states, memory_read_transformers(pm, addresses))
  AND
  transformer_invariant?(states, memory_write_transformers(pm, addresses))
  AND
  Forall(s : State, a : Address, b : Byte) :
    states(s) AND addresses(a) AND
    OK?(memory_write(pm)(a,b)(s)) AND
    OK?(memory_read(pm)(a)(state(memory_write(pm)(a,b)(s))))
```

```
      IMPLIES
        data(memory_read(pm)(a)(state(memory_write(pm)(a,b)(s)))) = b


  % proof status :-)
  changed_memory_invariant_mono : Lemma
    Forall(addresses_1, addresses_2 : PRED[Address]) :
      subset?(addresses_1, addresses_2) AND
      changed_memory_invariant?(pm, states, addresses_2)
      IMPLIES
        changed_memory_invariant?(pm, states, addresses_1)

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % unchanged results : memory_read_list
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  memory_read_list_nse_ok : Lemma
    Forall(raddr : Address, size : nat, s : State) :
      transformer_invariant?(states,
          memory_read_transformers(pm, address_block(raddr, size)))
      And
      transformers_ok?(states,
          memory_read_transformers(pm, address_block(raddr, size)))
      And
      states(s)
      Implies
          OK?(memory_read_list_nse(pm)(raddr, size)(s))

  % proof status :-)
  % memory_read_list_nse_next_length_TCC1 : TCC Obligation

  % proof status :-)
  % memory_read_list_nse_next_length_TCC2 : TCC Obligation

  % proof status :-)
  memory_read_list_nse_next_length : Lemma
    Forall(raddr : Address, size : nat, s : State) :
      transformer_invariant?(states,
                  memory_read_transformers(pm, address_block(raddr, size)))
      AND
      transformers_ok?(states,
                  memory_read_transformers(pm, address_block(raddr, size))) AND
      states(s)
      Implies
          states(state(memory_read_list_nse(pm)(raddr, size)(s)))
        And
          length(data(memory_read_list_nse(pm)(raddr, size)(s))) = size
```

```
% if read_memory is ok then read_list is ok
% proof status :-)
memory_read_list_ok : Lemma
  Forall(raddr : Address, size : nat, s : State) :
    transformer_invariant?(states,
      union(
        memory_read_transformers(pm, address_block(raddr, size)),
        memory_read_side_effect_super_transformers(pm,
                                        address_block(raddr, size))))
    And
    transformers_ok?(states,
      union(
        memory_read_transformers(pm, address_block(raddr, size)),
        memory_read_side_effect_super_transformers(pm,
                                        address_block(raddr, size))))
    And
    states(s)
    Implies
        OK?(memory_read_list(pm)(raddr, size)(s))

% proof status :-)
% memory_read_list_next_ok_TCC1 : TCC Obligation

% if read_memory is ok then next state of read_list is in states
% proof status :-)
memory_read_list_next_ok : Lemma
  Forall(raddr : Address, size : nat, s : State) :
    transformer_invariant?(states,
      union(
        memory_read_transformers(pm, address_block(raddr, size)),
        memory_read_side_effect_super_transformers(pm,
                                            address_block(raddr, size))))
    AND
    transformers_ok?(states,
      union(
        memory_read_transformers(pm, address_block(raddr, size)),
        memory_read_side_effect_super_transformers(pm,
                                            address_block(raddr, size))))
    And
    states(s)
    Implies
      states(state(memory_read_list(pm)(raddr, size)(s)))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read_list ## q
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% unchanged_memory_read_list_nse_TCC1 : TCC Obligation
```

```
% proof status :-)
% unchanged_memory_read_list_nse_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_nse_TCC3 : TCC Obligation

% The obvious unchanged lemma for memory_read_list
% proof status :-)
unchanged_memory_read_list_nse : Lemma
  Forall(raddr : Address, size : nat, s : State,
                        q : [State -> SuperResult[State]]) :
    unchanged_memory_invariant?(pm, states,
      union(memory_read_transformers(pm, address_block(raddr, size)),
            singleton(q)),
      addresses)
    AND
    transformers_ok?(states,
                     memory_read_transformers(pm, address_block(raddr, size)))
    AND
    subset?(address_block(raddr, size), addresses) AND
    states(s) AND
    OK?(q(s))
    IMPLIES
      data(memory_read_list_nse(pm)(raddr, size)(state(q(s)))) =
        data(memory_read_list_nse(pm)(raddr, size)(s))

% proof status :-)
% unchanged_memory_read_list_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_read_list_TCC3 : TCC Obligation

% proof status :-)
unchanged_memory_read_list : Lemma
  Forall(raddr : Address, size : nat, s : State,
                        q : [State -> SuperResult[State]]) :
    unchanged_memory_invariant?(pm, states,
      union(
        union(memory_read_transformers(pm, address_block(raddr, size)),
              memory_read_side_effect_super_transformers(pm,
                                                         address_block(raddr, size))),
        singleton(q)),
      addresses)
    AND
    transformers_ok?(states,
      union(
        memory_read_transformers(pm, address_block(raddr, size)),
        memory_read_side_effect_super_transformers(pm,
```

```
                                        address_block(raddr, size))))
    AND
    side_effect_content_unchanged(address_block(raddr, size), states,
                                  memory_read_side_effect(pm))
    And
    subset?(address_block(raddr, size), addresses) AND
    states(s) AND
    OK?(q(s))
    IMPLIES
      data(memory_read_list(pm)(raddr, size)(state(q(s)))) =
        data(memory_read_list(pm)(raddr, size)(s))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% memory_read ## memory_read_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
% unchanged_memory_invariant_unchanged_read_list_nse_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_read_list_nse_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_read_list_nse_TCC3 : TCC Obligation

% proof status :-)
unchanged_memory_invariant_unchanged_read_list_nse : Lemma
  Forall(s : State, addr_1, addr_2 : Address, size : nat) :
    unchanged_memory_invariant?(pm, states,
        memory_read_transformers(pm, address_block(addr_2, size)),
        addresses)
    AND
    transformers_ok?(states, memory_read_transformers(pm, addresses))
    AND
    subset?(address_block(addr_2, size), addresses)
    And
    addresses(addr_1)
    And
    states(s)
    IMPLIES
      data(memory_read(pm)(addr_1)(
                      state(memory_read_list_nse(pm)(addr_2, size)(s)))) =
      data(memory_read(pm)(addr_1)(s))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% normal termination of memory_write_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% if write_memory is ok then write_list is ok
% proof status :-)
memory_write_list_ok_nse : Lemma
  Forall(waddr : Address, s : State, bl : list[Byte]) :
    transformer_invariant?(states,
          memory_write_transformers(pm, address_block(waddr, length(bl))))
    AND
    transformers_ok?(states,
          memory_write_transformers(pm, address_block(waddr, length(bl))))
    AND
    states(s)
    IMPLIES
        OK?(memory_write_list_nse(pm)(waddr, bl)(s))


% proof status :-)
memory_write_list_ok : Lemma
  Forall(waddr : Address, s : State, bl : list[Byte]) :
    transformer_invariant?(states,
      union(memory_write_transformers(pm, address_block(waddr, length(bl))),
            memory_write_side_effect_super_transformers(pm,
                                                address_block(waddr, length(bl)))))
    AND
    transformers_ok?(states,
      union(memory_write_transformers(pm, address_block(waddr, length(bl))),
            memory_write_side_effect_super_transformers(pm,
                                                address_block(waddr, length(bl)))))
    AND
    side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                  memory_write_side_effect(pm))
    And
    states(s)
    IMPLIES
        OK?(memory_write_list(pm)(waddr, bl)(s))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unchanged results : memory_read_list ## memory_write_list
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_nse_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_nse_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_nse_TCC3 : TCC Obligation
```

```
% proof status :-)
unchanged_memory_invariant_unchanged_write_list_nse : Lemma
  Forall(s : State, raddr, waddr : Address, bl : list[Byte]) :
    unchanged_memory_invariant?(pm, states,
        memory_write_transformers(pm, address_block(waddr, length(bl))),
        singleton(raddr))
    AND
    transformers_ok?(states,
        memory_write_transformers(pm, address_block(waddr, length(bl))))
    AND
    transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr))))
    AND
    states(s)
    IMPLIES
      data(memory_read(pm)(raddr)(
        state(memory_write_list_nse(pm)(waddr, bl)(s)))) =
      data(memory_read(pm)(raddr)(s))


% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_TCC1 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_TCC2 : TCC Obligation

% proof status :-)
% unchanged_memory_invariant_unchanged_write_list_TCC3 : TCC Obligation

% proof status :-)
unchanged_memory_invariant_unchanged_write_list : Lemma
  Forall(s : State, raddr, waddr : Address, bl : list[Byte]) :
    unchanged_memory_invariant?(pm, states,
      union(memory_write_transformers(pm, address_block(waddr, length(bl))),
            memory_write_side_effect_super_transformers(pm,
                                                address_block(waddr, length(bl)))),
      singleton(raddr))
    AND
    transformers_ok?(states,
      union(memory_write_transformers(pm, address_block(waddr, length(bl))),
              memory_write_side_effect_super_transformers(pm,
                                                address_block(waddr, length(bl)))))
    AND
    transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr))))
    AND
    side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                  memory_write_side_effect(pm))
    AND
    states(s)
    IMPLIES
      data(memory_read(pm)(raddr)(
        state(memory_write_list(pm)(waddr, bl)(s)))) =
```

```
      data(memory_read(pm)(raddr)(s))

  % unchanged memory is an invariant of read_list
  % proof status :-)
  unchanged_memory_invariant_read_list : Lemma
    Forall(addr : Address, size : nat) :
      unchanged_memory_invariant?(pm, states, union(
        memory_read_transformers(pm, address_block(addr, size)),
        memory_read_side_effect_super_transformers(pm,
                                                   address_block(addr, size))),
        addresses)
      AND
      transformers_ok?(states,
        union(
          memory_read_transformers(pm, addresses),
          memory_read_side_effect_super_transformers(pm, addresses)))
      And
      subset?(address_block(addr, size), addresses)
      IMPLIES
        unchanged_memory_invariant?(pm, states,
            singleton(expr_2_super(memory_read_list(pm)(addr, size))),
            addresses)

  % proof status :-)
  % unchanged_memory_read_list_write_list_nse_TCC1 : TCC Obligation

  % proof status :-)
  % unchanged_memory_read_list_write_list_nse_TCC2 : TCC Obligation

  % proof status :-)
  unchanged_memory_read_list_write_list_nse : Lemma
    Forall(waddr : Address, bl : list[Byte], s : State) :
      unchanged_memory_invariant?(pm, states,
        memory_read_transformers(pm, address_block(waddr, length(bl))),
        address_block(waddr, length(bl)))
      AND
      unchanged_memory_write_invariant?(pm, states,
                                        address_block(waddr, length(bl)))
      AND
      changed_memory_invariant?(pm, states, address_block(waddr, length(bl)))
      AND
      transformers_ok?(states,
          memory_read_transformers(pm, address_block(waddr, length(bl))))
      AND
      transformers_ok?(states,
          memory_write_transformers(pm, address_block(waddr, length(bl))))
      AND
      states(s)
      IMPLIES
        data(memory_read_list_nse(pm)(waddr, length(bl))(
          state(memory_write_list_nse(pm)(waddr, bl)(s)))) = bl
```

```
  % proof status :-)
  % unchanged_memory_read_list_write_list_TCC1 : TCC Obligation

  % proof status :-)
  % unchanged_memory_read_list_write_list_TCC2 : TCC Obligation

  % proof status :-)
  unchanged_memory_read_list_write_list : Lemma
    Forall(waddr : Address, bl : list[Byte], s : State) :
      unchanged_memory_invariant?(pm, states, union(
        memory_read_transformers(pm, address_block(waddr, length(bl))),
        union(
          memory_read_side_effect_super_transformers(pm,
                                                     address_block(waddr, length(bl))),
          memory_write_side_effect_super_transformers(pm,
                                                     address_block(waddr, length(bl))))),
        address_block(waddr, length(bl)))
      AND
      unchanged_memory_write_invariant?(pm, states,
                                        address_block(waddr, length(bl)))
      AND
      changed_memory_invariant?(pm, states, address_block(waddr, length(bl)))
      AND
      transformers_ok?(states,
        union(
          memory_read_transformers(pm, address_block(waddr, length(bl))),
          memory_read_side_effect_super_transformers(pm,
                                                     address_block(waddr, length(bl)))))
      AND
      transformers_ok?(states,
        union(
          memory_write_transformers(pm, address_block(waddr, length(bl))),
          memory_write_side_effect_super_transformers(pm,
                                                     address_block(waddr, length(bl)))))
      AND
      side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                    memory_read_side_effect(pm))
      AND
      side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                    memory_write_side_effect(pm))
      AND
      states(s)
      IMPLIES
        data(memory_read_list(pm)(waddr, length(bl))(
          state(memory_write_list(pm)(waddr, bl)(s)))) = bl


End Memory_Change_2
```

B.19 paging-data-models.pvs

```
PDBR_Data_Model : Theory
Begin
  Importing Bits, PDBR_Type, Interpreted_Data[Pdbr_type]

% 31 12 5 4 3 2 0
% +-----------------------------------------------+
% | | |P|P| |
% | page dir base | empty |C|W| emp |
% | | |D|T| |
% +-----------------------------------------------+


  % proof status :-)
  % pdbr_valid?_TCC1 : TCC Obligation

  % proof status :-)
  % pdbr_valid?_TCC2 : TCC Obligation

  pdbr_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4 And
    % upper 20 bits are the base
    % next 7 bits are empty
    cut_bits(nth(data, 1), 0, 4) = 0 And
    cut_bits(nth(data, 0), 5, 3) = 0 And
    % bit 4 is PCD
    % bit 3 is PWT
    % remainder is empty
    cut_bits(nth(data, 0), 0, 3) = 0

  % proof status :-)
  % pdbr_to_byte_TCC1 : TCC Obligation

  % proof status :-)
  % pdbr_to_byte_TCC2 : TCC Obligation

  % proof status :-)
  % pdbr_to_byte_TCC3 : TCC Obligation

  pdbr_to_byte(pdbr : Pdbr_type, a : Address) : list[Byte] =
    (: overwrite_bool_bit(
                  overwrite_bool_bit(0, pdbr`pwt, 3),
                  pdbr`pcd, 4),
         cut_bits(offset(pdbr`base_addr), 8, 8),
         cut_bits(offset(pdbr`base_addr), 16, 8),
         cut_bits(offset(pdbr`base_addr), 24, 8)
    :)

  % proof status :-)
  % pdbr_from_byte_TCC1 : TCC Obligation

  % proof status :-)
  % pdbr_from_byte_TCC2 : TCC Obligation
```

```
  % proof status :-)
  % pdbr_from_byte_TCC3 : TCC Obligation

  % proof status :-)
  % pdbr_from_byte_TCC4 : TCC Obligation

  % proof status :-)
  % pdbr_from_byte_TCC5 : TCC Obligation

  pdbr_from_byte(data : list[Byte], a : Address) : lift[Pdbr_type] =
    IF pdbr_valid?(data, a) Then
      up((# base_addr :=
              Mem(
                   overwrite_shift_bits(
                     overwrite_shift_bits(
                       overwrite_shift_bits(0, nth(data, 1), 8, 8),
                     nth(data, 2), 16, 8),
                   nth(data, 3), 24, 8)
              ),
            pcd := cut_bit(nth(data, 0), 4),
            pwt := cut_bit(nth(data, 0), 3)
        #))
    Else bottom
    Endif

  % proof status :-)
  % pdbr_model_TCC1 : TCC Obligation

  pdbr_model : Interpreted_data_type[Pdbr_type] = (#
    uidt := (#
      size := 4,
      valid? := pdbr_valid?
    #),
    to_byte := pdbr_to_byte,
    to_mask := Lambda (d : Pdbr_type, a : Address) :
                  (zero_mask?(4)),
    from_byte := pdbr_from_byte
  #)


  % proof status :-)
  pdbr_valid_to_byte : Lemma Forall(d: Pdbr_type, a: Address):
    valid?(uidt(pdbr_model))(to_byte(pdbr_model)(d, a), a)


  % proof status :-)
  pdbr_is_pod : Theorem pod_data_type?(pdbr_model)

End PDBR_Data_Model
```

```
PDE_Model : Theory
Begin

  % PDE 4K
  %
  % 32 12 11 9 8 7 6 5 4 3 2 1 0
  % +-----------------------------------------------------------------
  % | | | | | R | | P | P | U | R | |
  % | page table address | avail | G | P | E | A | C | W | / | / | P |
  % | | | | S | S | | D | T | S | W | |
  % +-----------------------------------------------------------------
  %
  % PDE 4M
  %
  % 32 22 21 13 12 11 9 8 7 6 5 4 3 2 1 0
  % +-----------------------------------------------------------------
  % | | | P | | | | | | P | P | U | R | |
  % | page address | reserved | A | avail | G | P | D | A | C | W | / | / | P |
  % | | | T | | | S | | | D | T | S | W | |
  % +-----------------------------------------------------------------
  %
  % G : global page (ignored for 4K PTE's)
  % PS : page size (1 for superpages)
  % RES : reserved (0)
  % A : accessed
  % PCD : cache disabled
  % PWT : cache write through
  % U/S : user / supervisor
  % R/W : read / write
  % P : present
  % PAT : page table attribute index
  % D : dirty

  Importing Pde_type, Memory_privilege_util, Interpreted_Data

  % proof status :-)
  % pde_valid?_TCC1 : TCC Obligation

  % proof status :-)
  % pde_valid?_TCC2 : TCC Obligation

  % proof status :-)
  % pde_valid?_TCC3 : TCC Obligation

  pde_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4 And
    IF cut_bit(nth(data, 0), 0)
    Then % present
      IF cut_bit(nth(data, 0), 7)
      Then % superpage entry
        cut_bits(nth(data, 1), 5, 3) = 0 And
        cut_bits(nth(data, 2), 0, 6) = 0
```

```
    Else % PTE entry
      cut_bits(nth(data, 0), 6, 1) = 0
    Endif
  Else % not present
    True
  Endif


% proof status :-)
% pde_pte_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_pte_to_byte_TCC2 : TCC Obligation

% proof status :-)
% pde_pte_to_byte_TCC3 : TCC Obligation

pde_pte_to_byte(pde : Pde_pt_type, a : Address) : list[Byte] =
  (:
      overwrite_bool_bit(
        overwrite_bool_bit(
          overwrite_bool_bit(
            overwrite_bool_bit(
              overwrite_bool_bit(
                overwrite_bool_bit(0, true, 0),
                memory_access_to_bools(pde`access)`1, 1),
              memory_privilege_to_bool(pde`privilege), 2),
            pde`write_through, 3),
          pde`cache_disabled, 4),
        pde`accessed, 5),

      overwrite_shift_bits(
        overwrite_bool_bit(0, pde`global_page, 0),
        cut_bits(offset(pde`page_table_base), 12, 4),
        4, 4),

      cut_bits(offset(pde`page_table_base), 16, 8),
      cut_bits(offset(pde`page_table_base), 24, 8)
  :)

% proof status :-)
% pde_super_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_super_to_byte_TCC2 : TCC Obligation

% proof status :-)
% pde_super_to_byte_TCC3 : TCC Obligation

pde_super_to_byte(spage : Pde_4m_type, a : Address) : list[Byte] =
  (:
      overwrite_bool_bit(
```

```
          overwrite_bool_bit(
            overwrite_bool_bit(
              overwrite_bool_bit(
                overwrite_bool_bit(
                  overwrite_bool_bit(
                    overwrite_bool_bit(
                      overwrite_bool_bit(0, true, 0),
                      memory_access_to_bools(spage`access)`1, 1),
                    memory_privilege_to_bool(spage`privilege), 2),
                  spage`write_through, 3),
                spage`cache_disabled, 4),
              spage`accessed, 5),
            spage`dirty, 6),
          true, 7),

        overwrite_bool_bit(
          overwrite_bool_bit(0, spage`global_page, 0),
          spage`page_table_attribute_index, 4),

        shift_bits_left(cut_bits(offset(spage`page_base), 22, 2), 6),
        cut_bits(offset(spage`page_base), 24, 8)
    :)

  % proof status :-)
  % pde_to_byte_TCC1 : TCC Obligation

  % proof status :-)
  % pde_to_byte_TCC2 : TCC Obligation

  pde_to_byte(pde : Pde_type, a : Address) : list[Byte] =
    Cases pde of
      Not_present : (: 0, 0, 0, 0 :),
      Pde_pt(pte) : pde_pte_to_byte(pte, a),
      Pde_4m(spage) : pde_super_to_byte(spage, a)
    EndCases


  % proof status :-)
  % pde_4m_from_byte_TCC1 : TCC Obligation

  % proof status :-)
  % pde_4m_from_byte_TCC2 : TCC Obligation

  % proof status :-)
  % pde_4m_from_byte_TCC3 : TCC Obligation

  % proof status :-)
  % pde_4m_from_byte_TCC4 : TCC Obligation

  % proof status :-)
  % pde_4m_from_byte_TCC5 : TCC Obligation
```

```
% proof status :-)
% pde_4m_from_byte_TCC6 : TCC Obligation

pde_4m_from_byte(data : list[Byte], a : Address) : lift[Pde_4m_type] =
  % The first if here seems redundant. However, we do
  % down(x_pod'from_byte...) which only works on valid data. The
  % (worse) alternative is to restrict the input to valid data-address
  % pairs.
  IF pde_valid?(data, a) And
      cut_bit(nth(data, 0), 0) And
      cut_bit(nth(data, 0), 7)
  Then
    up((#
      page_base :=
        Mem(
              overwrite_shift_bits(
                overwrite_shift_bits(0, cut_bits(nth(data, 2), 6, 2), 22, 2),
              nth(data, 3), 24, 8)
         ),
      page_table_attribute_index := cut_bit(nth(data, 1), 4),
      global_page := cut_bit(nth(data, 1), 0),
      dirty := cut_bit(nth(data, 0), 6),
      accessed := cut_bit(nth(data, 0), 5),
      cache_disabled := cut_bit(nth(data, 0), 4),
      write_through := cut_bit(nth(data, 0), 3),
      privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
      access := bools_to_memory_access(cut_bit(nth(data, 0), 1), true)
    #))
  Else bottom
  Endif

% proof status :-)
% pde_4k_from_byte_TCC1 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC2 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC3 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC4 : TCC Obligation

% proof status :-)
% pde_4k_from_byte_TCC5 : TCC Obligation

pde_4k_from_byte(data : list[Byte], a : Address) : lift[Pde_pt_type] =
  % Again the first if seems redundant ...
  IF pde_valid?(data, a) And
      cut_bit(nth(data, 0), 0) And
      Not cut_bit(nth(data, 0), 7)
  Then
```

```
      up((#
        page_table_base :=
          Mem(
              overwrite_shift_bits(
                overwrite_shift_bits(
                  overwrite_shift_bits(0, cut_bits(nth(data, 1), 4, 4),
                  12, 4),
                nth(data, 2), 16, 8),
              nth(data, 3), 24, 8)
          ),
        global_page := cut_bit(nth(data, 1), 0),
        accessed := cut_bit(nth(data, 0), 5),
        cache_disabled := cut_bit(nth(data, 0), 4),
        write_through := cut_bit(nth(data, 0), 3),
        privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
        access := bools_to_memory_access(cut_bit(nth(data, 0), 1), true)
      #))
    Else bottom
    Endif

  % proof status :-)
  % pde_from_byte_TCC1 : TCC Obligation

  % proof status :-)
  % pde_from_byte_TCC2 : TCC Obligation

  pde_from_byte(data : list[Byte], a : Address) : lift[Pde_type] =
    IF pde_valid?(data, a) Then
      IF cut_bit(nth(data, 0), 0)
      Then % present
        IF cut_bit(nth(data, 0), 7)
        Then % super page entry
          up(Pde_4m(down(pde_4m_from_byte(data, a))))
        Else % page table entry
          up(Pde_pt(down(pde_4k_from_byte(data, a))))
        Endif
      Else % not present
        up(Not_present)
      Endif
    Else bottom
    Endif

  % proof status :-)
  % pde_model_TCC1 : TCC Obligation

  pde_model : Interpreted_data_type[Pde_type] = (#
    uidt := (#
      size := 4,
      valid? := pde_valid?
    #),
    to_byte := pde_to_byte,
    to_mask := Lambda (d : Pde_type, a : Address) :
```

```
                (zero_mask?(4)),
    from_byte := pde_from_byte
  #)

  % proof status :-)
  pde_valid_to_byte : Lemma Forall(d: Pde_type, a: Address):
    valid?(uidt(pde_model))(to_byte(pde_model)(d, a), a)

  % proof status :-)
  pde_is_pod : Theorem pod_data_type?(pde_model)

End PDE_Model


PTE_Model : Theory
Begin

  % PTE
  %
  % 32 12 11 9 8 7 6 5 4 3 2 1 0
  % +-----------------------------------------------------------------
  % | | | | P | | | P | P | U | R | |
  % | page address | avail | G | A | D | A | C | W | / | / | P |
  % | | | | T | | | D | T | S | W | |
  % +-----------------------------------------------------------------
  % G : global page
  % PAT : page table attribute index
  % D : dirty
  % A : accessed
  % PCD : cache disabled
  % PWT : cache write through
  % U/S : user / supervisor
  % R/W : read / write
  % P : present

  Importing Pte_type, Memory_privilege_util, Interpreted_Data

  pte_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4


  % proof status :-)
  % pte_page_to_byte_TCC1 : TCC Obligation

  % proof status :-)
  % pte_page_to_byte_TCC2 : TCC Obligation

  % proof status :-)
  % pte_page_to_byte_TCC3 : TCC Obligation

  pte_page_to_byte(pte : Pte_4k_type, a : Address) : list[Byte] =
```

```
  (:
      overwrite_bool_bit(
        overwrite_bool_bit(
          overwrite_bool_bit(
            overwrite_bool_bit(
              overwrite_bool_bit(
                overwrite_bool_bit(
                  overwrite_bool_bit(
                    overwrite_bool_bit(0, true, 0),
                    memory_access_to_bools(pte`access)`1, 1),
                  memory_privilege_to_bool(pte`privilege), 2),
                pte`write_through, 3),
              pte`cache_disabled, 4),
            pte`accessed, 5),
          pte`dirty, 6),
        pte`page_table_attribute_index, 7),

      overwrite_shift_bits(
        overwrite_bool_bit(0, pte`global_page, 0),
        cut_bits(offset(pte`page_base), 12, 4),
        4, 4),

      cut_bits(offset(pte`page_base), 16, 8),
      cut_bits(offset(pte`page_base), 24, 8)
  :)


% proof status :-)
% pte_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pte_to_byte_TCC2 : TCC Obligation

pte_to_byte(pte : Pte_type, a : Address) : list[Byte] =
  Cases pte of
    Not_present : (: 0, 0, 0, 0 :),
    Pte(pte) : pte_page_to_byte(pte, a)
  EndCases


% proof status :-)
% pte_page_from_byte_TCC1 : TCC Obligation

% proof status :-)
% pte_page_from_byte_TCC2 : TCC Obligation

% proof status :-)
% pte_page_from_byte_TCC3 : TCC Obligation

% proof status :-)
% pte_page_from_byte_TCC4 : TCC Obligation
```

```
% proof status :-)
% pte_page_from_byte_TCC5 : TCC Obligation

% proof status :-)
% pte_page_from_byte_TCC6 : TCC Obligation

pte_page_from_byte(data : list[Byte], a : Address) : lift[Pte_4k_type] =
  % The first if seems redundant ... see earlier comments on this
  IF pte_valid?(data, a) And cut_bit(nth(data, 0), 0)
  Then
    up((#
      page_base :=
        Mem(
            overwrite_shift_bits(
              overwrite_shift_bits(
                overwrite_shift_bits(0, cut_bits(nth(data, 1), 4, 4),
                12, 4),
              nth(data, 2), 16, 8),
            nth(data, 3), 24, 8)
        ),
      global_page := cut_bit(nth(data, 1), 0),
      page_table_attribute_index := cut_bit(nth(data, 0), 7),
      dirty := cut_bit(nth(data, 0), 6),
      accessed := cut_bit(nth(data, 0), 5),
      cache_disabled := cut_bit(nth(data, 0), 4),
      write_through := cut_bit(nth(data, 0), 3),
      privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
      access := bools_to_memory_access(cut_bit(nth(data, 0), 1), true)
    #))
  Else bottom
  Endif

% proof status :-)
% pte_from_byte_TCC1 : TCC Obligation

pte_from_byte(data : list[Byte], a : Address) : lift[Pte_type] =
  IF pte_valid?(data, a) Then
    IF cut_bit(nth(data, 0), 0)
    Then % present
      up(Pte(down(pte_page_from_byte(data, a))))
    Else % not present
      up(Not_present)
    Endif
  Else bottom
  Endif


% proof status :-)
% pte_model_TCC1 : TCC Obligation

pte_model : Interpreted_data_type[Pte_type] = (#
  uidt := (#
```

```
      size := 4,
      valid? := pte_valid?
    #),
    to_byte := pte_to_byte,
    to_mask := Lambda (d : Pte_type, a : Address) :
                 (zero_mask?(4)),
    from_byte := pte_from_byte
  #)


  % proof status :-)
  pte_valid_to_byte : Lemma Forall(d : Pte_type, a : Address) :
    pte_model`uidt`valid?(pte_model`to_byte(d, a), a)

  % proof status :-)
  pte_is_pod : Theorem pod_data_type?(pte_model)

End PTE_Model


PF_EC_Model : Theory
Begin

  Importing PF_Error_Code_Type, Memory_privilege_util

% Page Fault Error Code
% 31 5 4 3 2 1 0
% +-----------------------------------------------+
% | | I | R | U | W | |
% | Reserved | / | S | / | / | P |
% | | D | V | S | R | |
% +-----------------------------------------------+

  % proof status :-)
  % pf_ec_valid?_TCC1 : TCC Obligation

  % proof status :-)
  % pf_ec_valid?_TCC2 : TCC Obligation

  % proof status :-)
  % pf_ec_valid?_TCC3 : TCC Obligation

  % proof status :-)
  % pf_ec_valid?_TCC4 : TCC Obligation

  pf_ec_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4 And
    % upper 27 bits are reserved
    cut_bits(nth(data, 3), 0, 8) = 0 And
    cut_bits(nth(data, 2), 0, 8) = 0 And
    cut_bits(nth(data, 1), 0, 8) = 0 And
    cut_bits(nth(data, 0), 5, 3) = 0
```

```
  % bit 4 is instruction / data
  % bit 3 is reserved bit
  % bit 2 is user supervisor
  % bit 1 is write / read
  % bit 1 is present

% proof status :-)
% pf_ec_to_byte_TCC1 : TCC Obligation

% proof status :-)
% pf_ec_to_byte_TCC2 : TCC Obligation

pf_ec_to_byte(pf_ec : PF_error_code_type, a : Address) : list[Byte] =
  (: overwrite_bool_bit(
       overwrite_bool_bit(
         overwrite_bool_bit(
           overwrite_bool_bit(
             overwrite_bool_bit(0, pf_ec`present, 0),
             member(Write, pf_ec`access), 1),
           memory_privilege_to_bool(pf_ec`privilege), 2),
         pf_ec`reserved_bit_violation, 3),
       member(Execute, pf_ec`access), 4),
     0,
     0,
     0
  :)

% proof status :-)
% pf_ec_from_byte_TCC1 : TCC Obligation

% proof status :-)
% pf_ec_from_byte_TCC2 : TCC Obligation

pf_ec_from_byte(data : list[Byte], a : Address) : lift[PF_error_code_type] =
  If pf_ec_valid?(data, a) Then
    up((# reserved_bit_violation := cut_bit(nth(data, 0), 3),
          privilege := bool_to_memory_privilege(cut_bit(nth(data, 0), 2)),
          access := bools_to_memory_access(cut_bit(nth(data, 0), 1), cut_bit(nth(data, 0), 4)),
          present := cut_bit(nth(data, 0), 0)
       #))
  Else
    bottom
  Endif

% proof status :-)
% pf_ec_model_TCC1 : TCC Obligation

pf_ec_model : Interpreted_data_type[PF_error_code_type] = (#
  uidt := (#
    size := 4,
    valid? := pf_ec_valid?
  #),
```

```
    to_byte := pf_ec_to_byte,
    to_mask := Lambda (d : PF_error_code_type, a : Address) :
                 (zero_mask?(4)),
    from_byte := pf_ec_from_byte
  #)


  % proof status :-)
  pf_ec_valid_to_byte : Lemma Forall(d: PF_error_code_type, a: Address):
    valid?(uidt(pf_ec_model))(to_byte(pf_ec_model)(d, a), a)

  % proof status :-)
  pf_ec_is_pod : Theorem pod_data_type?(pf_ec_model)

End PF_EC_Model


Address_Model : Theory
Begin

  Importing Address_Datatype

  address_valid?(data : list[Byte], a : Address) : bool =
    length(data) = 4

  % proof status :-)
  % address_to_byte_TCC1 : TCC Obligation

  % proof status :-)
  % address_to_byte_TCC2 : TCC Obligation

  % proof status :-)
  % address_to_byte_TCC3 : TCC Obligation

  address_to_byte(addr : Memory_Address_4G, a : Address) : list[Byte] =
    (: cut_bits(offset(addr), 0, 8),
       cut_bits(offset(addr), 8, 8),
       cut_bits(offset(addr), 16, 8),
       cut_bits(offset(addr), 24, 8)
    :)

  % proof status :-)
  % address_from_byte_TCC1 : TCC Obligation

  % proof status :-)
  % address_from_byte_TCC2 : TCC Obligation

  % proof status :-)
  % address_from_byte_TCC3 : TCC Obligation

  % proof status :-)
  % address_from_byte_TCC4 : TCC Obligation
```

```
  % proof status :-)
  % address_from_byte_TCC5 : TCC Obligation

  address_from_byte(data : list[Byte], a : Address) : lift[Memory_Address_4G] =
    If address_valid?(data, a) Then
      up(Mem(overwrite_shift_bits(
                overwrite_shift_bits(
                  overwrite_shift_bits(
                    overwrite_shift_bits(0,
                    nth(data, 0), 0, 8),
                  nth(data, 1), 8, 8),
                nth(data, 2), 16, 8),
              nth(data, 3), 24, 8)))
    Else
      bottom
    Endif

  % proof status :-)
  % address_model_TCC1 : TCC Obligation

  address_model : Interpreted_data_type[Memory_Address_4G] = (#
    uidt := (#
      size := 4,
      valid? := address_valid?
    #),
    to_byte := address_to_byte,
    to_mask := Lambda (d : Memory_Address_4G, a : Address) :
                  (zero_mask?(4)),
    from_byte := address_from_byte
  #)


  % proof status :-)
  address_valid_to_byte : Lemma Forall(d: Memory_Address_4G, a: Address):
    valid?(uidt(address_model))(to_byte(address_model)(d, a), a)


  % proof status :-)
  address_is_pod : Theorem pod_data_type?(address_model)

End Address_Model
```

B.20 paging-data.pvs

```
PDBR_Type : Theory
Begin
  Importing IA32, Alignment

  % proof status :-)
  % Pdbr_type_TCC1 : TCC Obligation
```

```
  Pdbr_type : Type = [#
    base_addr : {a : Memory_Address_4G | aligned?(12)(offset(a))},
    pcd : bool,
    pwt : bool
  #]

End PDBR_Type


PDBR_Datatype : Theory
% axioms are discharched in PDBR_Data_Model
Begin
  Importing PDBR_Type, Interpreted_Data[Pdbr_type]

  pdbr_data_type_exists : Axiom EXISTS(x: (pod_data_type?[Pdbr_type])): TRUE

  % proof status :-)
  % pdbr_data_type_TCC1 : TCC Obligation

  pdbr_data_type : (pod_data_type?[Pdbr_type])

End PDBR_Datatype


% Page_Directory_Types[T : Type] : Theory
% type dependency waits until available bits are used
Page_Directory_Types : Theory
Begin
  Importing IA32, Memory_privilege, Memory_access_util, Alignment

  % proof status :-)
  % Pde_pt_type_TCC1 : TCC Obligation

  Pde_pt_type : Type = [#
    page_table_base : {a : Memory_Address_4G | aligned?(12)(offset(a))},
    % pde_4k_avail : T,
    global_page : bool,
    accessed : bool,
    cache_disabled : bool,
    write_through : bool,
    privilege : Memory_privilege,
    access : RX_Memory_access
  #]

  Pde_4m_type : Type = [#
    page_base : {a : Memory_Address_4G | aligned?(22)(offset(a))},
    page_table_attribute_index : bool,
    % pde_4m_avail : T,
    global_page : bool,
    dirty : bool,
    accessed : bool,
```

```
    cache_disabled : bool,
    write_through : bool,
    privilege : Memory_privilege,
    access : RX_Memory_access
  #]

  Pte_4k_type : Type = [#
    page_base : {a : Memory_Address_4G | aligned?(12)(offset(a))},
    % pte_avail : T,
    global_page : bool,
    page_table_attribute_index : bool,
    dirty : bool,
    accessed : bool,
    cache_disabled : bool,
    write_through : bool,
    privilege : Memory_privilege,
    access : RX_Memory_access
  #]

End Page_Directory_Types


%Pde_type[X, Y : Type] : Datatype
% type dependency waits until available bits are used
Pde_type : Datatype
Begin
  Importing Page_Directory_Types

  % Not_present(data : Y) : not_present?
  Not_present : not_present?
  Pde_pt(pde_pt : Pde_pt_type) : pde_pt?
  Pde_4m(pde_4m : Pde_4m_type) : pde_4m?
End Pde_type


PDE_Datatype : Theory
Begin
  Importing Pde_type, Interpreted_Data

  pde_data_type_exists : Axiom
    Exists(x : (pod_data_type?[Pde_type])) : True

  % proof status :-)
  % pde_data_type_TCC1 : TCC Obligation

  pde_data_type : (pod_data_type?[Pde_type])

End PDE_Datatype


Pte_type : Datatype
Begin
```

```
  Importing Page_Directory_Types

  Not_present : not_present?
  Pte(pte : Pte_4k_type) : pte?
End Pte_type


PTE_Datatype : Theory
Begin
  Importing Pte_type, Interpreted_Data

  pte_data_type_exists : Axiom
        Exists(x : (pod_data_type?[Pte_type])) : True

  % proof status :-)
  % pte_data_type_TCC1 : TCC Obligation

  pte_data_type : (pod_data_type?[Pte_type])

End PTE_Datatype

%%
% Use this type only in linear resolve and only after verifying the type
%
Paging_type : Datatype
Begin
  Importing Pde_type, Pte_type

  Ptab(ptab : Pte_type) : ptab?
  Pdir(pdir : Pde_type) : pdir?

End Paging_type

Paging_type_helpers : Theory
Begin
  Importing Paging_type

  pt : Var Paging_type

  max_level : nat = 2
  Level : Type = below[max_level]

  % proof status :-)
  % range_pt_TCC1 : TCC Obligation

  range_pt(lvl : Level) : PRED[Paging_type] =
    Lambda (p : Paging_type) :
      Cond
        lvl = 0 -> pdir?(p),
        lvl = 1 -> ptab?(p)
      EndCond
```

```
set_accessed(pt) : Paging_type =
  Cases pt Of
    Ptab(ptab) :
      Cases ptab Of
        Not_present : Ptab(Not_present),
        Pte(pe) : Ptab(Pte(pe With [accessed := true]))
      EndCases,
    Pdir(pdir) :
      Cases pdir Of
        Not_present : Pdir(Not_present),
        Pde_pt(pe) : Pdir(Pde_pt(pe With [accessed := true])),
        Pde_4m(pe) : Pdir(Pde_4m(pe With [accessed := true]))
      EndCases
  EndCases

set_dirty(pt) : Paging_type =
  Cases pt Of
    Ptab(ptab) :
      Cases ptab Of
        Not_present : Ptab(Not_present),
        Pte(pe) : Ptab(Pte(pe With [dirty := true]))
      EndCases,
    Pdir(pdir) :
      Cases pdir Of
        Not_present : Pdir(Not_present),
        Pde_pt(pe) : Pdir(Pde_pt(pe)),
        Pde_4m(pe) : Pdir(Pde_4m(pe With [dirty := true]))
      EndCases
  EndCases

set_reference(pt : Paging_type, access : Memory_access) : Paging_type =
  If Write?(access) Then
    set_accessed(set_dirty(pt))
  Else
    set_accessed(pt)
  Endif

present?(pt) : bool =
  Cases pt Of
    Ptab(ptab) : Not not_present?(ptab),
    Pdir(pdir) : Not not_present?(pdir)
  EndCases

% proof status :-)
% base_TCC1 : TCC Obligation

% proof status :-)
% base_TCC2 : TCC Obligation

base(pt : (present?)) : Memory_Address_4G =
  Cases pt Of
    Ptab(ptab) : page_base(pte(ptab)),
```

```
      Pdir(pdir) :
        Cases pdir Of
          Pde_pt(pe) : page_table_base(pe),
          Pde_4m(pe) : page_base(pe)
        EndCases
    EndCases

  accessible?(pt : (present?), access : Memory_access) : bool =
    Cases pt Of
      Ptab(ptab) : member(access, access(pte(ptab))),
      Pdir(pdir) :
        Cases pdir Of
          Pde_pt(pe) : member(access, access(pe)),
          Pde_4m(pe) : member(access, access(pe))
        EndCases
    EndCases

  privileged?(pt : (present?), priv : Memory_privilege) : bool =
    Cases pt Of
      Ptab(ptab) : Not Supervisor?(priv) And User?(privilege(pte(ptab))),
      Pdir(pdir) :
        Cases pdir Of
          Pde_pt(pe) : Not Supervisor?(priv) And User?(privilege(pe)),
          Pde_4m(pe) : Not Supervisor?(priv) And User?(privilege(pe))
        EndCases
    EndCases

  is_leaf?(pt : (present?)) : bool =
    Not (pdir?(pt) And pde_pt?(pdir(pt)))

  pdir_entry?(pt) : bool =
    pdir?(pt)

  ptab_entry?(pt) : bool =
    ptab?(pt)

  paging_type?(lvl : Level, pe : Paging_type) : bool =
    Cond
      lvl = 0 -> pdir_entry?(pe),
      lvl = 1 -> ptab_entry?(pe)
    EndCond

  % proof status :-)
  set_reference_range : Lemma Forall (lvl : Level, pt : (range_pt(lvl)), access : Memory_access) :
    range_pt(lvl)(set_reference(pt, access))

  % proof status :-)
  set_reference_pdir_entry : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
    set_reference(pt1, access) = set_reference(pt2, access) Implies
      pdir_entry?(pt1) = pdir_entry?(pt2)

  % proof status :-)
```

```
  set_reference_ptab_entry : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
    set_reference(pt1, access) = set_reference(pt2, access) Implies
      ptab_entry?(pt1) = ptab_entry?(pt2)

  % proof status :-)
  set_reference_paging_type : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access, lvl : Level) :
    set_reference(pt1, access) = set_reference(pt2, access) Implies
      paging_type?(lvl, pt1) = paging_type?(lvl, pt2)

  % proof status :-)
  set_reference_present : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
    set_reference(pt1, access) = set_reference(pt2, access) Implies
      present?(pt1) = present?(pt2)

  % proof status :-)
  % set_reference_base_TCC1 : TCC Obligation

  % proof status :-)
  set_reference_base : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
    set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
      base(pt1) = base(pt2)

  % proof status :-)
  set_reference_accessible : Lemma Forall (pt1, pt2 : Paging_type, access, ac1 : Memory_access) :
    set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
      accessible?(pt1, ac1) = accessible?(pt2, ac1)

  % proof status :-)
  set_reference_privileged : Lemma
    Forall (pt1, pt2 : Paging_type, access : Memory_access, priv : Memory_privilege) :
      set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
        privileged?(pt1, priv) = privileged?(pt2, priv)

  % proof status :-)
  set_reference_leaf : Lemma Forall (pt1, pt2 : Paging_type, access : Memory_access) :
    set_reference(pt1, access) = set_reference(pt2, access) And present?(pt1) Implies
      is_leaf?(pt1) = is_leaf?(pt2)

End Paging_type_helpers


% To do: remove this theory
Paging_Datatype : Theory
Begin

  Importing Paging_type_helpers,
            PTE_Datatype, PDE_Datatype, Interpreted_Data_Lift

  % proof status :-)
  % paging_data_type_TCC1 : TCC Obligation

  % proof status :-)
```

```
  % paging_data_type_TCC2 : TCC Obligation

  % proof status :-)
  % paging_data_type_TCC3 : TCC Obligation

  % proof status :-)
  % paging_data_type_TCC4 : TCC Obligation

  % proof status :-)
  % paging_data_type_TCC5 : TCC Obligation

  % proof status :-)
  % paging_data_type_TCC6 : TCC Obligation

  % proof status :-)
  % paging_data_type_TCC7 : TCC Obligation

  paging_data_type(lvl : Level) : (pod_data_type?[(range_pt(lvl))]) =
    Cond
      lvl = 0 -> dt_lift[Pde_type, (range_pt(lvl))](pde_data_type,
                   restrict[(pdir?), (range_pt(lvl)), Pde_type](pdir), Pdir),
      lvl = 1 -> dt_lift[Pte_type, (range_pt(lvl))](pte_data_type,
                   restrict[(ptab?), (range_pt(lvl)), Pte_type](ptab), Ptab)
    EndCond

  pe_size : nat = 2 % log2 size of page table entries

  paging_data_type_size : Axiom
    Forall (lvl : Level) :
      size(uidt(paging_data_type(lvl))) = expt(2, pe_size)

End Paging_Datatype


PF_Error_Code_Type : Theory
Begin
  Importing Interpreted_Data, Error_Code_Types

  pf_ec_type_exists : Axiom
    Exists(x : (pod_data_type?[PF_error_code_type])) : True

  % proof status :-)
  % pf_ec_data_type_TCC1 : TCC Obligation

  pf_ec_data_type : (pod_data_type?[PF_error_code_type])

End PF_Error_Code_Type


Address_Datatype : Theory
Begin
```

```
  Importing Interpreted_Data

  address_data_type_exists : Axiom
    Exists(x : (pod_data_type?[Memory_Address_4G])) : True

  % proof status :-)
  % address_data_type_TCC1 : TCC Obligation

  address_data_type : (pod_data_type?[Memory_Address_4G])

End Address_Datatype


Segment_Types : Theory
Begin

  Importing Memory_privilege, IA32

  Segment_type : Type = {Data_RO_Accessed, % 0001
                         Data_RW_Accessed, % 0011
                         Code_EO_Accessed, % 1001
                         Code_ER_Accessed, % 1011
                         % System
                         LDT, % 0010
                         TSS32_Avail, % 1001
                         Interrupt_Gate, % 1110
                         Trap_Gate % 1111
                        }

  Segment_Selector_type : Type = [#
    index : below[expt(2,13)],
    table_indicator : bool, % GDT = false, LDT = true
    requested_privilege_level : Memory_privilege
  #]

  Access_Rights_type : Type = [#
    segment_type : Segment_type,
    system : bool,
    privilege : Memory_privilege,
    present : bool,
    % bits 11:8 reserved
    avail : bool,
    long_mode : bool,
    op_size_32 : bool, % true = 32 bit, false = 16 bit
    gran : bool,
    unusable : bool
    % bits 31:7 reserved
  #]

  % In the VMCS, IA32 stores the elements of this record in separate fields,
  % Only the selector field is architecturally visible. The remaining
  % fields are shadow fields.
```

```
  Segment_Reg_type : Type = [#
    selector : Segment_Selector_type,
    base_address : Memory_Address_4G,
    limit : below[expt(2,32)],
    access_rights : Access_Rights_type
  #]

  segment_to_priv(cs : Segment_Reg_type) : Memory_privilege =
    cs`access_rights`privilege

End Segment_Types


Segment_Reg_Datatype : Theory
Begin

  Importing Interpreted_Data, Segment_Types

  % Segment Selector

  segment_selector_type_exists : Axiom
    Exists (x : (pod_data_type?[Segment_Selector_type])) : True

  % proof status :-)
  % segment_selector_data_type_TCC1 : TCC Obligation

  segment_selector_data_type : (pod_data_type?[Segment_Selector_type])

  % Segment Register

  segment_reg_data_type_exists : Axiom
    Exists(x : (pod_data_type?[Segment_Reg_type])) : True

  % proof status :-)
  % segment_reg_data_type_TCC1 : TCC Obligation

  segment_reg_data_type : (pod_data_type?[Segment_Reg_type])

End Segment_Reg_Datatype


Control_Register_Types : Theory
Begin

  Importing IA32

  CR0_type : Type = [#
    foo : bool,
    bar : bool
  #]

  CR2_type : Type = Memory_Address_4G
```

```
  CR4_type : Type = [#
    foo : bool
  #]

End Control_Register_Types

Control_Register_Datatype : Theory
Begin

  Importing Control_Register_Types, Interpreted_Data

  cr0_data_type_exists : Axiom
    Exists(x : (pod_data_type?[CR0_type])) : True

  % proof status :-)
  % cr0_data_type_TCC1 : TCC Obligation

  cr0_data_type : (pod_data_type?[CR0_type])

  cr2_data_type_exists : Axiom
    Exists(x : (pod_data_type?[CR2_type])) : True

  % proof status :-)
  % cr2_data_type_TCC1 : TCC Obligation

  cr2_data_type : (pod_data_type?[CR2_type])

  cr4_data_type_exists : Axiom
    Exists(x : (pod_data_type?[CR4_type])) : True

  % proof status :-)
  % cr4_data_type_TCC1 : TCC Obligation

  cr4_data_type : (pod_data_type?[CR4_type])

End Control_Register_Datatype
```

B.21 physical_memory.pvs

```
% specification of physical memory


Physical_Memory [(Importing IA32) min, max : Memory_Address] : Theory
Begin

  IMPORTING ExprResult, Unit

  Physical_memory : Type = [Address -> Byte]

  % read & write
```

```
  s : Var Physical_memory
  a : Var Address
  b : Var Byte
  bl : Var list[Byte]
  cp : Var bool % cross page access

  physical_read_side_effect(id : Address, bl, cp)(s) : ExprResult[Physical_memory, list[Byte]] =
    %Cases id Of
    %Else
      OK(s, bl)
    %EndCases

  physical_write_side_effect(id : Address, bl, cp)(s) : ExprResult[Physical_memory, list[Byte]] =
    %Cases id Of
    %Else
      OK(s, bl)
    %EndCases

  physical_read(a)(s) : ExprResult[Physical_memory, Byte] =
    If in_memory(min, max)(a) Then
      OK(s, s(a))
    Else
      Fatal
    Endif

  physical_write(a : Address, b : Byte)(s) : ExprResult[Physical_memory, Unit] =
    If in_memory(min, max)(a) Then
      OK(s With [(a) := b], unit)
    Else
      Fatal
    Endif

End Physical_Memory


Physical_Memory_Corollaries [(Importing IA32) min, max : Memory_Address ] : Theory
Begin

  IMPORTING Physical_Memory[min, max]

  s : Var Physical_memory
  a : Var Address
  b : Var Byte

  %%%%%% Next states

  % proof status :-)
  % physical_read_ok_next_state_TCC1 : TCC Obligation

  % proof status :-)
  physical_read_ok_next_state : Lemma
    OK?(physical_read(a)(s)) Implies state(physical_read(a)(s)) = s
```

```
  % proof status :-)
  % physical_write_ok_next_state_TCC1 : TCC Obligation

  % proof status :-)
  physical_write_ok_next_state : Lemma
    OK?(physical_write(a, b)(s)) Implies
      state(physical_write(a, b)(s)) = s with [(a) := b]

  %%%%%% Returned data

  % proof status :-)
  physical_read_ok_get_data : Lemma
    OK?(physical_read(a)(s)) Implies
      data(physical_read(a)(s)) = s(a)


End Physical_Memory_Corollaries


Physical_Memory_Properties [(Importing IA32) min, max : Memory_Address ] : Theory
Begin

  IMPORTING Physical_Memory[min, max]

  s : Var Physical_memory
  a : Var Address
  b : Var Byte

  %%%%%%%%%%%%%%%%%%%%%%%%%%%% utility lemma

  % proof status :-)
  physical_read_ok : Lemma in_memory(min, max)(a) Implies OK?(physical_read(a)(s))

  % proof status :-)
  physical_write_ok : Lemma in_memory(min, max)(a) Implies OK?(physical_write(a, b)(s))


End Physical_Memory_Properties


Memory_Physical_Memory[(IMPORTING IA32) min, max : Memory_Address] : Theory
% Defines the physical memory instance of memory
Begin
  IMPORTING Physical_Memory[min, max]
  IMPORTING Memory[Physical_memory]
%% IMPORTING Transformer_Result_State

  %%%%% Public Interface %%%%%

  phy_mem : Memory_struct = (#
```

```
    memory_read := physical_read,
    memory_write := physical_write,

    memory_write_side_effect := physical_write_side_effect,
    memory_read_side_effect := physical_read_side_effect
  #)

End Memory_Physical_Memory
```

B.22 plain_memory.pvs

```
Plain_Memory[ State : Type ] : Theory
% plain memory definition and immediate utility results
Begin

  IMPORTING Memory_Change_2

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % Plain Memory Definition
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  Plain_Memory : Type = [#
    mem : Memory_struct[State],
    states : PRED[State],
    ro_addr : PRED[Address],
    rw_addr : PRED[Address],
    other_actions : PRED[[State -> SuperResult[State]]]
  #]

  pm : Var Plain_Memory

  addr : Var Address
  size : Var nat


% We need to add one additional set of addresses to express that
% certain device state remains unchanged despite blessed accesses.
% Thereby the memory itself needs not to be blessed.
%
% Example: writing to RAM does not change the APIC EOI register state.
%
% Read, write, other_action and side effect transformers must establish
% an invariant which states:
% - that when reading a subset of the addresses is allowed, the same
% values are read, and
% - that device accesses behave the same before and after the blessed
% memory access
% - This rules out that the memory access counter is in the above set
% which is what we desire.
% - How can we formalize this point? How do we prevent the device from
```

```
% changing internal state which may only be seen after a sequence of
% future accesses?

  plain_memory?(pm) : bool =
    % there is an invariant hidden, see below
    unchanged_memory_invariant?(pm`mem, pm`states,
      union(
        union(pm`other_actions,
          memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr))),
        union(memory_read_side_effect_super_transformers(pm`mem,
                                                union(pm`ro_addr, pm`rw_addr)),
              memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))),
      union(pm`ro_addr, pm`rw_addr))
    AND
    unchanged_memory_invariant?(pm`mem, pm`states,
                  memory_write_transformers(pm`mem, pm`rw_addr),
                  pm`ro_addr)
    AND
    unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)
    AND
    changed_memory_invariant?(pm`mem, pm`states, pm`rw_addr)
    AND
    transformers_ok?(pm`states,
      union(
        union(memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
              memory_write_transformers(pm`mem, pm`rw_addr)),
        union(memory_read_side_effect_super_transformers(pm`mem,
                                                union(pm`ro_addr, pm`rw_addr)),
              memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))))
    And
    side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                  memory_read_side_effect(pm`mem))
    And
    side_effect_content_unchanged(pm`rw_addr, pm`states,
                                  memory_write_side_effect(pm`mem))

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % Utility results
  %
  % transformer invariants
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  plain_memory_invariant_read_transformers_ro_addr : Lemma
    plain_memory?(pm) Implies
      transformer_invariant?(pm`states,
                             memory_read_transformers(pm`mem, pm`ro_addr))

  % proof status :-)
```

```
plain_memory_invariant_read_transformers_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm`states,
                           memory_read_transformers(pm`mem, pm`rw_addr))

% proof status :-)
plain_memory_invariant_read_transformers_ro_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm`states,
      memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)))

% proof status :-)
plain_memory_invariant_write_transformers_rw_addr : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm`states,
                           memory_write_transformers(pm`mem, pm`rw_addr))


% proof status :-)
plain_memory_invariant : Lemma
  plain_memory?(pm) Implies
    transformer_invariant?(pm`states,
      union(
        pm`other_actions,
        union(union(memory_read_transformers(pm`mem,
                                             union(pm`ro_addr, pm`rw_addr)),
                    memory_write_transformers(pm`mem, pm`rw_addr)),
              union(memory_read_side_effect_super_transformers(pm`mem,
                                             union(pm`ro_addr, pm`rw_addr)),
                    memory_write_side_effect_super_transformers(pm`mem,
                                             pm`rw_addr)))))

% proof status :-)
plain_memory_unchanged_invariant : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_invariant?(pm`mem, pm`states,
      union(
        union(pm`other_actions,
          memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr))),
        union(memory_read_side_effect_super_transformers(pm`mem,
                                               union(pm`ro_addr, pm`rw_addr)),
            memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))),
        union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_unchanged_memory_invariant_write : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_invariant?(pm`mem, pm`states,
              memory_write_transformers(pm`mem, pm`rw_addr),
              pm`ro_addr)

% proof status :-)
```

```
plain_memory_unchanged_memory_write_invariant : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)

% proof status :-)
plain_memory_changed_memory_invariant : Lemma
  plain_memory?(pm) Implies
    changed_memory_invariant?(pm`mem, pm`states, pm`rw_addr)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% transformer invariants on blocks
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_invariant_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      transformer_invariant?(pm`states, memory_read_transformers(pm`mem,
                                                address_block(addr, size)))

% proof status :-)
plain_memory_invariant_write_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      transformer_invariant?(pm`states,
              memory_write_transformers(pm`mem, address_block(addr, size)))

% proof status :-)
plain_memory_transformer_invariant_read_side_effects : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      transformer_invariant?(pm`states,
        memory_read_side_effect_super_transformers(pm`mem,
                                                address_block(addr, size)))

% proof status :-)
plain_memory_transformer_invariant_write_side_effects : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      transformer_invariant?(pm`states,
        memory_write_side_effect_super_transformers(pm`mem,
                                                address_block(addr, size)))

% proof status :-)
plain_memory_transformer_invariant_read_list_block : Lemma
```

```
    plain_memory?(pm) And
    subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
      Implies
        transformer_invariant?(pm`states,
          union(memory_read_transformers(pm`mem, address_block(addr, size)),
                memory_read_side_effect_super_transformers(pm`mem,
                                                           address_block(addr, size))))


  % proof status :-)
  plain_memory_transformer_invariant_write_list_block : Lemma
    plain_memory?(pm) And
    subset?(address_block(addr, size), pm`rw_addr)
      Implies
        transformer_invariant?(pm`states,
          union(memory_write_transformers(pm`mem, address_block(addr, size)),
                memory_write_side_effect_super_transformers(pm`mem,
                                                            address_block(addr, size))))



  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % transformers_ok?
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  plain_memory_transformers_ok_read_ro_rw : Lemma
    plain_memory?(pm)
      Implies
        transformers_ok?(pm`states,
            memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)))

  % proof status :-)
  plain_memory_transformers_ok_read_block : Lemma
    plain_memory?(pm) And
    subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
      Implies
        transformers_ok?(pm`states, memory_read_transformers(pm`mem,
                                                    address_block(addr, size)))

  % proof status :-)
  plain_memory_transformers_ok_write_rw : Lemma
    plain_memory?(pm)
      Implies
        transformers_ok?(pm`states,
            memory_write_transformers(pm`mem, pm`rw_addr))

  % proof status :-)
  plain_memory_transformers_ok_write_block : Lemma
    plain_memory?(pm) And
```

```
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      transformers_ok?(pm`states,
              memory_write_transformers(pm`mem, address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_read_side_effects_ro_rw : Lemma
  plain_memory?(pm)
    Implies
      transformers_ok?(pm`states,
        memory_read_side_effect_super_transformers(pm`mem,
                                                   union(pm`ro_addr, pm`rw_addr)))

% proof status :-)
plain_memory_transformers_ok_read_side_effects_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      transformers_ok?(pm`states,
        memory_read_side_effect_super_transformers(pm`mem,
                                                   address_block(addr, size)))

% proof status :-)
plain_memory_transformers_ok_write_side_effects_ro_rw : Lemma
  plain_memory?(pm)
    Implies
      transformers_ok?(pm`states,
        memory_write_side_effect_super_transformers(pm`mem, pm`rw_addr))

% proof status :-)
plain_memory_transformers_ok_write_side_effects_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      transformers_ok?(pm`states,
        memory_write_side_effect_super_transformers(pm`mem,
                                                   address_block(addr, size)))



% proof status :-)
plain_memory_transformers_ok_read_list_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      transformers_ok?(pm`states,
        union(memory_read_transformers(pm`mem, address_block(addr, size)),
              memory_read_side_effect_super_transformers(pm`mem,
                                                         address_block(addr, size))))


% proof status :-)
```

```
plain_memory_transformers_ok_write_list_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      transformers_ok?(pm`states,
        union(memory_write_transformers(pm`mem, address_block(addr, size)),
              memory_write_side_effect_super_transformers(pm`mem,
                                                  address_block(addr, size))))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unchanged_memory_invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_unchanged_read_block_rw_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_transformers(pm`mem, address_block(addr, size)),
              pm`rw_addr)

% proof status :-)
plain_memory_unchanged_read_ro_rw_addr : Lemma
  plain_memory?(pm) Implies
    unchanged_memory_invariant?(pm`mem, pm`states,
            memory_read_transformers(pm`mem, union(pm`ro_addr, pm`rw_addr)),
            union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_unchanged_read_block_ro_rw_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_transformers(pm`mem, address_block(addr, size)),
              union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_unchanged_write_block_ro_addr : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_write_transformers(pm`mem, address_block(addr, size)),
              pm`ro_addr)

% unchanged invariants of reading
% proof status :-)
```

```
plain_memory_unchanged_read_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_transformers(pm`mem, address_block(addr, size)),
              address_block(addr, size))

% proof status :-)
plain_memory_unchanged_read_side_effect_block_ro_rw : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_side_effect_super_transformers(pm`mem,
                                                  address_block(addr, size)),
              union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_unchanged_read_side_effect_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_read_side_effect_super_transformers(pm`mem,
                                                    address_block(addr, size)),
              address_block(addr, size))

% proof status :-)
plain_memory_unchanged_write_side_effect_block_ro_rw : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_write_side_effect_super_transformers(pm`mem,
                                                    address_block(addr, size)),
              union(pm`ro_addr, pm`rw_addr))

% proof status :-)
plain_memory_unchanged_write_side_effect_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      unchanged_memory_invariant?(pm`mem, pm`states,
              memory_write_side_effect_super_transformers(pm`mem,
                                                    address_block(addr, size)),
              address_block(addr, size))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% unchanged_memory_write_invariant
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_unchanged_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      unchanged_memory_write_invariant?(pm`mem, pm`states,
                                                address_block(addr, size))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% changed_memory_invariant
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_changed_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
      changed_memory_invariant?(pm`mem, pm`states, address_block(addr, size))

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% side_effect_content_unchanged
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_side_effect_content_unchanged_read_block : Lemma
 plain_memory?(pm) And
 subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
   Implies
 side_effect_content_unchanged(address_block(addr, size), pm`states,
                                  memory_read_side_effect(pm`mem))


% proof status :-)
plain_memory_side_effect_content_unchanged_write_block : Lemma
  plain_memory?(pm) And
  subset?(address_block(addr, size), pm`rw_addr)
    Implies
  side_effect_content_unchanged(address_block(addr, size), pm`states,
                                   memory_write_side_effect(pm`mem))


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% single reads/writes
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% proof status :-)
plain_memory_memory_read_ok : Lemma Forall(s : State) :
  plain_memory?(pm) And
  pm`states(s) And
  union(pm`ro_addr, pm`rw_addr)(addr) Implies
    OK?(memory_read(pm`mem)(addr)(s))

% proof status :-)
% plain_memory_states_memory_read_TCC1 : TCC Obligation

% proof status :-)
plain_memory_states_memory_read : Lemma Forall(s : State) :
  plain_memory?(pm) And
  pm`states(s) And
  union(pm`ro_addr, pm`rw_addr)(addr) Implies
    pm`states(state(memory_read(pm`mem)(addr)(s)))

% proof status :-)
% plain_memory_memory_read_memory_read_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_memory_read_memory_read_TCC2 : TCC Obligation

% proof status :-)
% plain_memory_memory_read_memory_read_TCC3 : TCC Obligation

% proof status :-)
plain_memory_memory_read_memory_read : Lemma
  Forall(s : State, addr1, addr2 : Address) :
    plain_memory?(pm) And
    pm`states(s) And
    union(pm`ro_addr, pm`rw_addr)(addr1) And
    union(pm`ro_addr, pm`rw_addr)(addr2) Implies
      data(memory_read(pm`mem)(addr1)(state(memory_read(pm`mem)(addr2)(s)))) =
      data(memory_read(pm`mem)(addr1)(s))

% proof status :-)
plain_memory_memory_write_ok : Lemma Forall(s : State, byte : Byte) :
  plain_memory?(pm) And
  pm`states(s) And
  pm`rw_addr(addr) Implies
    OK?(memory_write(pm`mem)(addr, byte)(s))

% proof status :-)
% plain_memory_states_memory_write_TCC1 : TCC Obligation

% proof status :-)
plain_memory_states_memory_write : Lemma Forall(s : State, byte : Byte) :
  plain_memory?(pm) And
  pm`states(s) And
```

```
    pm‘rw_addr(addr) Implies
      pm‘states(state(memory_write(pm‘mem)(addr, byte)(s)))

  % proof status :-)
  % plain_memory_memory_read_memory_write_other_TCC1 : TCC Obligation

  % proof status :-)
  % plain_memory_memory_read_memory_write_other_TCC2 : TCC Obligation

  % proof status :-)
  % plain_memory_memory_read_memory_write_other_TCC3 : TCC Obligation

  % proof status :-)
  plain_memory_memory_read_memory_write_other : Lemma
    Forall(s : State, ro_addr, rw_addr : Address, byte : Byte) :
      plain_memory?(pm) And
      pm‘states(s) And
      union(pm‘ro_addr, pm‘rw_addr)(ro_addr) And
      pm‘rw_addr(rw_addr) And
      Not ro_addr = rw_addr Implies
        data(memory_read(pm‘mem)(ro_addr)
                          (state(memory_write(pm‘mem)(rw_addr, byte)(s)))) =
        data(memory_read(pm‘mem)(ro_addr)(s))


  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  %
  % restricting changes a la separation logic
  %
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  % proof status :-)
  % stays_unchanged_TCC1 : TCC Obligation

  % proof status :-)
  % stays_unchanged_TCC2 : TCC Obligation

  stays_unchanged(pm : (plain_memory?))
                 (s1, s2 : (pm‘states), addresses : PRED[Address]) : bool =
    Forall(addr : Address) :
      union(pm‘ro_addr, pm‘rw_addr)(addr) And
      addresses(addr) Implies
        data(memory_read(pm‘mem)(addr)(s1)) =
                        data(memory_read(pm‘mem)(addr)(s2))

  % proof status :-)
  % stays_unchanged_unchanged_TCC1 : TCC Obligation

  % proof status :-)
  % stays_unchanged_unchanged_TCC2 : TCC Obligation
```

```
% proof status :-)
stays_unchanged_unchanged : Lemma
  Forall(s1, s2 : State, addresses : PRED[Address], addr : Address) :
    plain_memory?(pm) And
    pm`states(s1) And pm`states(s2) And
    union(pm`ro_addr, pm`rw_addr)(addr) And
    stays_unchanged(pm)(s1, s2, addresses) And
    addresses(addr)
      Implies
        data(memory_read(pm`mem)(addr)(s1)) =
                        data(memory_read(pm`mem)(addr)(s2))

% proof status :-)
stays_unchanged_symmetric : Lemma
  Forall(s1, s2 : State, addresses : PRED[Address]) :
    plain_memory?(pm) And
    pm`states(s1) And pm`states(s2) Implies
      stays_unchanged(pm)(s1, s2, addresses) =
                        stays_unchanged(pm)(s2, s1, addresses)

% proof status :-)
stays_unchanged_mono : Lemma
  Forall(s1, s2 : State, addresses_1, addresses_2 : PRED[Address]) :
    plain_memory?(pm) And
    pm`states(s1) And pm`states(s2) And
    subset?(addresses_1, addresses_2) And
    stays_unchanged(pm)(s1, s2, addresses_2) Implies
      stays_unchanged(pm)(s1, s2, addresses_1)



only_changes(pm : (plain_memory?))
                (s1, s2 : (pm`states), addresses : PRED[Address]) : bool =
  stays_unchanged(pm)(s1, s2,
      union(pm`ro_addr, difference(pm`rw_addr, addresses)))

% proof status :-)
stays_unchanged_only_changes : Lemma
  Forall(s1, s2 : State, addresses : PRED[Address]) :
    plain_memory?(pm) And
    pm`states(s1) And pm`states(s2) And
    subset?(pm`ro_addr, addresses) Implies
      stays_unchanged(pm)(s1, s2, addresses) =
        only_changes(pm)(s1, s2, difference(pm`rw_addr, addresses))


% proof status :-)
stays_unchanged_only_changes_disjoint : Lemma
  Forall(s1, s2 : State, addresses_1, addresses_2 : PRED[Address]) :
    plain_memory?(pm) And
    pm`states(s1) And pm`states(s2) And
    subset?(addresses_2, union(pm`ro_addr, pm`rw_addr)) And
```

```
      disjoint?(addresses_1, addresses_2) And
      only_changes(pm)(s1, s2, addresses_1)
        Implies
          stays_unchanged(pm)(s1, s2, addresses_2)


  % proof status :-)
  only_changes_symmetric : Lemma
    Forall(s1, s2 : State, addresses : PRED[Address]) :
      plain_memory?(pm) And
      pm`states(s1) And pm`states(s2) Implies
        only_changes(pm)(s1, s2, addresses) =
                          only_changes(pm)(s2, s1, addresses)

  % proof status :-)
  only_changes_mono : Lemma
    Forall(s1, s2 : State, addresses_1, addresses_2 : PRED[Address]) :
      plain_memory?(pm) And
      pm`states(s1) And pm`states(s2) And
      subset?(addresses_1, addresses_2) And
      only_changes(pm)(s1, s2, addresses_1) Implies
        only_changes(pm)(s1, s2, addresses_2)

  % proof status :-)
  % only_changes_unchanged_TCC1 : TCC Obligation

  % proof status :-)
  % only_changes_unchanged_TCC2 : TCC Obligation

  % proof status :-)
  only_changes_unchanged : Lemma
    Forall(s1, s2 : State, addresses : PRED[Address]) :
      plain_memory?(pm) And
      pm`states(s1) And pm`states(s2) And
      only_changes(pm)(s1, s2, addresses) And
      (pm`ro_addr(addr) OR
         pm`rw_addr(addr) And Not addresses(addr))
      Implies
        data(memory_read(pm`mem)(addr)(s1)) =
                        data(memory_read(pm`mem)(addr)(s2))

  % proof status :-)
  % plain_memory_stays_unchanged_memory_read_TCC1 : TCC Obligation

  % proof status :-)
  plain_memory_stays_unchanged_memory_read : Lemma
    Forall(s : State, addr : Address) :
      plain_memory?(pm) And
      pm`states(s) And
      union(pm`ro_addr, pm`rw_addr)(addr)
        Implies
          stays_unchanged(pm)(s, state(memory_read(pm`mem)(addr)(s)),
```

```
                                        union(pm`ro_addr, pm`rw_addr))

  % proof status :-)
  % plain_memory_stays_unchanged_memory_write_TCC1 : TCC Obligation

  % proof status :-)
  plain_memory_stays_unchanged_memory_write : Lemma
    Forall(s : State, addr : Address, byte : Byte) :
      plain_memory?(pm) And
      pm`states(s) And
      pm`rw_addr(addr)
        Implies
          stays_unchanged(pm)(s,
                state(memory_write(pm`mem)(addr, byte)(s)),
                      remove(addr, union(pm`ro_addr, pm`rw_addr)))

End Plain_Memory


Abstract_Read_Write_Plain [State, Data : Type] : Theory
Begin

  Importing Plain_Memory, Abstract_Read_Write

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])
  s : Var State
  addr : Var Address
  data : Var Data

  write_data(pm, dt)(addr, data) : [State -> ExprResult[State, Unit]] =
    write_data(pm`mem, dt)(addr, data)

  read_data(pm, dt)(addr) : [State -> ExprResult[State, Data]] =
    read_data(pm`mem, dt)(addr)

  valid_in_mem(pm, dt)(addr) : [State -> bool] =
    valid_in_mem(pm`mem, dt)(addr)

End Abstract_Read_Write_Plain


Plain_Mem_Properties[State : Type] : Theory
%
% primitive (non-datatype) read and writes on plain memory
Begin

  IMPORTING Plain_Memory[State]

  pm : Var Plain_Memory[State]
```

```
  s : Var State
  addr : Var Address
  size : Var nat

  % reading a block of memory is a transformer invariant

  % writing to a block of memory is a transformer invariant

  % proof status :-)
  % plain_mem_write_list_read_list_TCC1 : TCC Obligation

  % proof status :-)
  plain_mem_write_list_read_list : Lemma
    Forall(size : nat, bl : list[Byte]) :
      size = length(bl) AND
      plain_memory?(pm) And
      pm`states(s) And
      subset?(address_block(addr, size), pm`rw_addr) And
      OK?(memory_write_list(pm`mem)(addr, bl)(s)) And
      OK?(memory_read_list(pm`mem)(addr, size)
          (state(memory_write_list(pm`mem) (addr, bl)(s))))
        Implies
          data(memory_read_list(pm`mem)(addr, size)
                   (state(memory_write_list (pm`mem) (addr, bl) (s))))
            = bl

  % proof status :-)
  % plain_mem_q_read_list_TCC1 : TCC Obligation

  % proof status :-)
  plain_mem_q_read_list : Lemma Forall(q : [State -> SuperResult[State]]) :
    OK?(memory_read_list(pm`mem)(addr, size)(s)) And
    OK?(q(s)) And
    OK?(memory_read_list(pm`mem)(addr, size)(state(q(s)))) And
    plain_memory?(pm) And
    unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                                address_block(addr, size)) And
    pm`states(s) And
    subset?(address_block(addr, size), union(pm`ro_addr, pm`rw_addr))
      Implies
        data(memory_read_list(pm`mem)(addr, size) (state(q(s))))
          = data(memory_read_list(pm`mem)(addr, size) (s))

End Plain_Mem_Properties


Plain_Mem_Properties_2[State, Data : Type] : Theory
Begin
  IMPORTING Plain_Mem_Properties, Abstract_Read_Write_Plain

  pm : Var Plain_Memory[State]
```

```
dt : Var (interpreted_data_type?[Data])
s : Var State
addr : Var Address
data : Var Data

% proof status :-)
% plain_memory_inv_pred_write_data_TCC1 : TCC Obligation

% proof status :-)
plain_memory_inv_pred_write_data : Lemma
  plain_memory?(pm) And
  pm`states(s) And
  subset?(address_block(addr, size(uidt(dt))), pm`rw_addr) And
  OK?(write_data(pm, dt)(addr, data)(s))
    Implies
      pm`states(state(write_data(pm, dt)(addr, data)(s)))

% proof status :-)
plain_memory_read_data_ok : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) And
  valid_in_mem(pm, dt)(addr)(s)
    Implies
      OK?(read_data(pm,dt)(addr)(s))


% proof status :-)
plain_memory_write_data_ok : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, pm`rw_addr)
    Implies
      OK?(write_data(pm,dt)(addr, data)(s))


% proof status :-)
% plain_memory_write_data_valid_TCC1 : TCC Obligation

% proof status :-)
plain_memory_write_data_valid : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt, addr, pm`rw_addr)
    Implies
      valid_in_mem(pm, dt)(addr)
              (state(write_data(pm, dt)(addr, data)(s)))


% proof status :-)
plain_memory_transformers_ok_write_data : Lemma
  plain_memory?(pm) AND
```

```
    in_blessed_memory?(dt, addr, pm`rw_addr)
      Implies
        transformers_ok?(pm`states,
                singleton(expr_2_super(write_data(pm, dt) (addr, data))))


  % proof status :-)
  plain_memory_transformer_invariant_write_data : Lemma
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, pm`rw_addr)
      Implies
        transformer_invariant?(pm`states,
                singleton(expr_2_super(write_data(pm, dt) (addr, data))))

  % proof status :-)
  plain_memory_transformer_invariant_read_data : Lemma
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr))
      Implies
        transformer_invariant?(pm`states,
                        singleton(expr_2_super(read_data(pm, dt) (addr))))


  % proof status :-)
  plain_memory_unchanged_memory_invariant_write_data : Lemma
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, pm`rw_addr)
      IMPLIES
        unchanged_memory_invariant?(pm`mem, pm`states,
            singleton(expr_2_super(write_data(pm, dt)(addr, data))),
            difference(union(pm`ro_addr, pm`rw_addr),
                                address_block(addr, size(uidt(dt)))))

  % proof status :-)
  plain_memory_unchanged_memory_invariant_read_data : Lemma
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr))
      IMPLIES
        unchanged_memory_invariant?(pm`mem, pm`states,
            singleton(expr_2_super(read_data(pm, dt)(addr))),
            union(pm`ro_addr, pm`rw_addr))

  % proof status :-)
  % plain_memory_read_data_q_ok_TCC1 : TCC Obligation

  % proof status :-)
  plain_memory_read_data_q_ok : Lemma
    Forall(q : [State -> SuperResult[State]]) :
      plain_memory?(pm) AND
      in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) AND
      unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                                  address_block(addr, size(uidt(dt)))) And
```

```
    pm‘states(s) And
    OK?(q(s)) And
    valid_in_mem(pm,dt)(addr)(s) Implies
      OK?(read_data(pm, dt)(addr)(state(q(s))))

% proof status :−)
% plain_memory_read_data_q_same_TCC1 : TCC Obligation

% proof status :−)
% plain_memory_read_data_q_same_TCC2 : TCC Obligation

% proof status :−)
plain_memory_read_data_q_same : Lemma
  Forall(q : [State −> SuperResult[State]]) :
    plain_memory?(pm) AND
    in_blessed_memory?(dt, addr, union(pm‘ro_addr, pm‘rw_addr)) AND
    unchanged_memory_invariant?(pm‘mem, pm‘states, singleton(q),
                                address_block(addr, size(uidt(dt)))) And
    pm‘states(s) And
    OK?(q(s)) And
    valid_in_mem(pm,dt)(addr)(s) Implies
      data(read_data(pm, dt)(addr)(state(q(s)))) =
        data(read_data(pm, dt)(addr)(s))


% proof status :−)
plain_memory_read_write_ok : Lemma
  plain_memory?(pm) AND
  pm‘states(s) And
  in_blessed_memory?(dt, addr, pm‘rw_addr)
    IMPLIES
      OK?((
            write_data(pm,dt)(addr, data) ##
            read_data(pm,dt)(addr)
          )(s)
         )

% proof status :−)
% plain_memory_read_write_res_TCC1 : TCC Obligation

% proof status :−)
plain_memory_read_write_res : Lemma
  plain_memory?(pm) AND
  pm‘states(s) And
  in_blessed_memory?(dt, addr, pm‘rw_addr)
    IMPLIES
      data((
             write_data(pm,dt)(addr, data) ##
             read_data(pm,dt)(addr)
           )(s))
        = data
```

```
  addresses : VAR PRED[Address]

  % proof status :-)
  plain_memory_unchanged_memory_ok_result : Lemma
    Forall (d : Data) :
      plain_memory?(pm) And
      subset?(addresses, union(pm`ro_addr, pm`rw_addr))
        Implies
          unchanged_memory_invariant?(pm`mem, pm`states,
                        singleton(expr_2_super[State, Data](ok_result(d))),
                        addresses)

end Plain_Mem_Properties_2

Plain_Mem_Properties_3[State, Data1, Data2 : Type] : Theory
%
% reads after write at different places return the previous item
Begin
  Importing Plain_Mem_Properties_2, Memory_Change_4
            % Memory_Change_To_Unit

  pm : Var Plain_Memory[State]

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  s : Var State
  addr1, addr2 : Var Address
  data1 : Var Data1


  % proof status :-)
  plain_memory_read_write_other_ok : Lemma
    plain_memory?(pm) AND
    pm`states(s) And
    in_blessed_memory?(dt1, addr1, pm`rw_addr) And
    in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
    valid_in_mem(pm,dt2)(addr2)(s)
      IMPLIES
        OK?
           ((
                write_data(pm,dt1)(addr1, data1) ##
                read_data(pm,dt2)(addr2)
            )(s)
           )


  % proof status :-)
  % plain_memory_read_write_other_res_TCC1 : TCC Obligation

  % proof status :-)
  % plain_memory_read_write_other_res_TCC2 : TCC Obligation
```

```
% proof status :-)
plain_memory_read_write_other_res : Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) AND
  valid_in_mem(pm,dt2)(addr2)(s)
    IMPLIES
      data((
                  write_data(pm,dt1)(addr1, data1) ##
                  read_data(pm,dt2)(addr2)
                )(s))
        =
      data(read_data(pm,dt2)(addr2)(s))

% proof status :-)
plain_memory_read_read_ok: Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  valid_in_mem(pm,dt1)(addr1)(s) And
  valid_in_mem(pm,dt2)(addr2)(s)
    IMPLIES
      OK?(
                ( read_data(pm, dt1)(addr1) ##
                  read_data(pm, dt2)(addr2)
                )(s)
        )

% proof status :-)
% plain_memory_read_read_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_read_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_read: Lemma
  plain_memory?(pm) AND
  pm`states(s) And
  in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) AND
  in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) AND
  valid_in_mem(pm,dt1)(addr1)(s) And
  valid_in_mem(pm,dt2)(addr2)(s)
    IMPLIES
      data(
                ( read_data(pm, dt1)(addr1) ##
                  read_data(pm, dt2)(addr2)
                )(s)
              )
```

```
        = data(read_data(pm, dt2)(addr2)(s))

  q : Var [State -> ExprResult[State, Data1]]
  r : Var [Data1 -> [State -> ExprResult[State, Data2]]]
  addresses : VAR PRED[Address]

  % proof status :-)
  plain_memory_unchanged_composition : Lemma
    Forall (P : PRED[Data1]) :
      plain_memory?(pm) And
      subset?(addresses, union(pm`ro_addr, pm`rw_addr)) And
      (Forall (s : (pm`states)) : OK?(q(s)) Implies P(data(q(s)))) And
      unchanged_memory_invariant?(pm`mem, pm`states,
                                  singleton(expr_2_super(q)), addresses)
      And
      (Forall (d : (P)) : unchanged_memory_invariant?(pm`mem, pm`states,
                                  singleton(expr_2_super(r(d))), addresses))
        Implies
          unchanged_memory_invariant?(pm`mem, pm`states,
                                  singleton(expr_2_super(q ## r)), addresses)

End Plain_Mem_Properties_3
```

B.23 plain_memory_rewrites.pvs

```
% Rewrite rules
%===============

Block_Disjoint_Rewrites : Theory
Begin

  Importing Address_Util

  addr1, addr2 : Var Address
  sz1, sz2 : Var nat

  % different registers are block disjoint
  %---------------------------------------

  % proof status :-)
  different_registers_blocks_disjoint : Lemma
    type_of(addr1) /= type_of(addr2)
      Implies
        blocks_disjoint?(addr1, sz1, addr2, sz2)

End Block_Disjoint_Rewrites


Plain_Mem_Rewrites1[State : Type] : Theory
Begin

  Importing Plain_Mem_Properties_3, ExpressionStatements
```

```
  pm : Var Plain_Memory[State]
  s : Var State

  % proof status :-)
  % pm_q_prop_TCC1 : TCC Obligation

  pm_q_prop(pm : Plain_Memory[State])(res : StmtResult[State]) : bool =
    OK?(res) And pm`states(state(res))

  % proof status :-)
  pm_q_prop_ok_stmt : Lemma % !!! Lemma does not rewrite
    Forall (q : [State -> StmtResult[State]]):
      plain_memory?(pm) And
      pm_q_prop(pm)(q(s))
    Implies
      OK?(q(s))


End Plain_Mem_Rewrites1


Plain_Mem_Rewrites2[State, Data : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites1

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])
  s : Var State
  addr : Var Address
  data : Var Data

  % in blessed memory rewrite
  % proof status :-)
  in_blessed_memory_rw_ro : Lemma % (P: 4)
    in_blessed_memory?(dt, addr, pm`rw_addr)
      Implies
        in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr))

  % pm_q_prop
  %------------

  % proof status :-)
  % pm_q_prop_TCC1 : TCC Obligation

  pm_q_prop(pm : Plain_Memory[State])(res : ExprResult[State, Data]) : bool =
    OK?(res) And pm`states(state(res))

  % proof status :-)
  pm_q_prop_ok_expr : Lemma % !!! Lemma does not rewrite
```

```
    Forall (q : [State -> ExprResult[State, Data]]) :
      plain_memory?(pm) And
      pm_q_prop(pm)(q(s))
    Implies
      OK?(q(s))

  % pm_q_prop simplification
  %-------------------------

  % proof status :-)
  pm_q_prop_single_read : Lemma %(P: 4)
      plain_memory?(pm) And
      pm`states(s) And
      in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) And
      valid_in_mem(pm, dt)(addr)(s)
    Implies
      pm_q_prop(pm)(read_data(pm, dt)(addr)(s))

  % proof status :-)
  pm_q_prop_ok_result_single : Lemma % (P: 4)
      pm`states(s)
    Implies
      pm_q_prop(pm)(ok_result(data)(s))

  % plain_write_data_q_ok
  %----------------------

  % proof status :-)
  plain_memory_write_ok_single : Lemma
      plain_memory?(pm) And
      pm`states(s) And
      in_blessed_memory?(dt, addr, pm`rw_addr)
    Implies
      OK?(write_data(pm, dt)(addr, data)(s))

  % proof status :-)
  % pm_q_prop_read_TCC1 : TCC Obligation

  pm_q_prop_read(pm : Plain_Memory[State], dt, addr)(res : StmtResult[State]) : bool =
    OK?(res) And OK?(read_data(pm, dt)(addr)(state(res))) And
    pm`states(state(res))

End Plain_Mem_Rewrites2

Plain_Mem_Rewrites3[State, Data, Data_q : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites2

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])
```

```
  s : Var State
  addr : Var Address

  % proof status :-)
  % pm_q_prop_read_TCC1 : TCC Obligation

  pm_q_prop_read(pm : Plain_Memory[State], dt, addr)(res : ExprResult[State, Data_q]) : bool =
    OK?(res) And OK?(read_data(pm, dt)(addr)(state(res))) And
    pm`states(state(res))

End Plain_Mem_Rewrites3


% Rewrite rules for lemmas from Plain_Mem_Properties_3
Plain_Mem_Rewrites4[State, Data, Data_q : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites3

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])
  s : Var State
  addr : Var Address
  data : Var Data

  % pm_q_prop
  %-----------

  % proof status :-)
  plain_memory_write_ok_q_expr : Lemma
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm_q_prop(pm)(q(s))
    Implies
      OK?((q ## write_data(pm, dt)(addr, data))(s))

  % proof status :-)
  pm_q_prop_read_ok_expr : Lemma
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      pm_q_prop_read(pm, dt, addr)(q(s))
    Implies
      OK?((q ## read_data(pm, dt)(addr))(s))

  % proof status :-)
  pm_q_prop_read_pm_q_prop_expr : Lemma
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) And
      pm_q_prop_read(pm, dt, addr)(q(s))
```

```
    Implies
      pm_q_prop(pm)((q ## read_data(pm, dt)(addr))(s))

  % proof status :-)
  pm_q_prop_read_write_q_expr : Lemma
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm_q_prop(pm)(q(s))
    Implies
      pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))

  % proof status :-)
  pm_q_prop_ok_result_q_expr : Lemma
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      pm_q_prop(pm)(q(s))
    Implies
      pm_q_prop(pm)((q ## ok_result[State, Data](data))(s))

  data_q : Var Data_q

  % proof status :-)
  pm_q_prop_read_ok_result_single : Lemma
      plain_memory?(pm) And
      pm`states(s) And
      in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) And
      valid_in_mem(pm, dt)(addr)(s)
    Implies
      pm_q_prop_read(pm, dt, addr)(ok_result[State, Data_q](data_q)(s))

  % data
  %------

  % proof status :-)
  % plain_memory_read_write_q_data_expr_TCC1 : TCC Obligation

  % proof status :-)
  plain_memory_read_write_q_data_expr : Lemma
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm_q_prop(pm)(q(s))
    Implies
      data((q ## write_data(pm, dt)(addr, data) ##
                                        read_data(pm, dt)(addr))(s)) = data

  % pm_q_prop simplification
  %-------------------------

  dt1 : Var (interpreted_data_type?[Data])
  dt2 : Var (interpreted_data_type?[Data_q])
```

```
  addr1 : Var Address
  addr2 : Var Address

  % proof status :-)
  pm_q_prop_read_write_other_single : Lemma %(P: 4)
    plain_memory?(pm) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    in_blessed_memory?(dt1, addr1, pm`rw_addr) And
    in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
    valid_in_mem(pm, dt2)(addr2)(s) And
    pm`states(s)
  Implies
    pm_q_prop_read(pm, dt2, addr2)(write_data(pm, dt1)(addr1, data)(s))

  % proof status :-)
  pm_q_prop_read_read_single : Lemma %(P: 4)
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
    valid_in_mem(pm, dt1)(addr1)(s) And
    valid_in_mem(pm, dt2)(addr2)(s) And
    pm`states(s)
  Implies
    pm_q_prop_read(pm, dt2, addr2)(read_data(pm, dt1)(addr1)(s))

End Plain_Mem_Rewrites4

Plain_Mem_Rewrites5[State, Data : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites4

  pm : Var Plain_Memory[State]

  dt : Var (interpreted_data_type?[Data])
  s : Var State
  addr : Var Address
  data : Var Data

  % pm_q_prop simplification
  %-------------------------

  % proof status :-)
  plain_memory_write_ok_q_stmt : Lemma
    Forall (q : [State -> StmtResult[State]]):
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm_q_prop(pm)(q(s))
    Implies
      OK?((q ## write_data(pm, dt)(addr, data))(s))
```

```
  % proof status :-)
  pm_q_prop_read_ok_stmt : Lemma
    Forall (q : [State -> StmtResult[State]]) :
      plain_memory?(pm) And
      pm_q_prop_read(pm, dt, addr)(q(s))
    Implies
      OK?((q ## read_data(pm, dt)(addr))(s))

  % proof status :-)
  pm_q_prop_read_pm_q_prop_stmt : Lemma
    Forall (q : [State -> StmtResult[State]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, union(pm`ro_addr, pm`rw_addr)) And
      pm_q_prop_read(pm, dt, addr)(q(s))
    Implies
      pm_q_prop(pm)((q ## read_data(pm, dt)(addr))(s))

  % proof status :-)
  pm_q_prop_read_write_q_stmt : Lemma
    Forall (q : [State -> StmtResult[State]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm_q_prop(pm)(q(s))
    Implies
      pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))

  % proof status :-)
  pm_q_prop_ok_result_q_stmt : Lemma
    Forall (q : [State -> StmtResult[State]]) :
      plain_memory?(pm) And
      pm_q_prop(pm)(q(s))
    Implies
      pm_q_prop(pm)((q ## ok_result[State, Data](data))(s))

  % proof status :-)
  % plain_memory_read_write_q_data_stmt_TCC1 : TCC Obligation

  % proof status :-)
  plain_memory_read_write_q_data_stmt : Lemma
    Forall (q : [State -> StmtResult[State]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm_q_prop(pm)(q(s))
    Implies
      data((q ## write_data(pm, dt)(addr, data) ##
                                        read_data(pm, dt)(addr))(s)) = data

  % proof status :-)
  pm_q_prop_single_write : Lemma
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm`states(s)
```

```
    Implies
      pm_q_prop(pm)(write_data(pm, dt)(addr, data)(s))

  % proof status :-)
  pm_q_prop_read_write_single : Lemma
      plain_memory?(pm) And
      in_blessed_memory?(dt, addr, pm`rw_addr) And
      pm`states(s)
    Implies
      pm_q_prop_read(pm, dt, addr)(write_data(pm, dt)(addr, data)(s))

  % plain_memory_read_write_single_data = plain_memory_read_write_res

  expr : VAR [State -> ExprResult[State, Data]]
  stmt : VAR [State -> StmtResult[State]]

  % proof status :-)
  pm_q_prop_e2s : Lemma
    pm_q_prop(pm)(e2s(expr)(s)) = pm_q_prop(pm)(expr(s))

  % proof status :-)
  pm_q_prop_stmt_e2s : Lemma
    pm_q_prop(pm)(stmt(s)) Implies
      pm_q_prop(pm)((stmt ## e2s(expr))(s)) = pm_q_prop(pm)((stmt ## expr)(s))

End Plain_Mem_Rewrites5


Plain_Mem_Rewrites6[State, Data1, Data2, Data_q : Type] : Theory

Begin

  Importing Plain_Mem_Rewrites5

  pm : Var Plain_Memory[State]

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  s : Var State
  addr1, addr2 : Var Address
  data1 : Var Data1
  data2 : Var Data2

  % pm_q_prop simplification
  %-------------------------

  % proof status :-)
  pm_q_prop_read_write_other_q_expr : Lemma %(P: 6)
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
      in_blessed_memory?(dt2, addr2, pm`rw_addr) And
```

```
      in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
      pm_q_prop_read(pm, dt1, addr1)(q(s))
    Implies
      pm_q_prop_read(pm, dt1, addr1)((q ## write_data(pm, dt2)(addr2, data2))(s))

  % proof status :-)
  pm_q_prop_read_read_q_expr : Lemma %(P: 6)
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
      in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
      pm_q_prop_read(pm, dt1, addr1)(q(s)) And
      pm_q_prop_read(pm, dt2, addr2)(q(s))
    Implies
      pm_q_prop_read(pm, dt2, addr2)((q ## read_data(pm, dt1)(addr1))(s))

  % proof status :-)
  pm_q_prop_read_ok_result_q_expr : Lemma % (P: 4)
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      pm_q_prop_read(pm, dt2, addr2)(q(s))
    Implies
      pm_q_prop_read(pm, dt2, addr2)((q ## ok_result[State, Data1](data1))(s))

  % plain_memory_read_read_single_data = plain_memory_read_read

  % proof status :-)
  % plain_memory_read_read_q_data_expr_TCC1 : TCC Obligation

  % proof status :-)
  % plain_memory_read_read_q_data_expr_TCC2 : TCC Obligation

  % proof status :-)
  plain_memory_read_read_q_data_expr : Lemma %(P: 6)
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
      in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
      pm_q_prop_read(pm, dt2, addr2)(q(s)) And
      pm_q_prop_read(pm, dt1, addr1)(q(s))
    Implies
      data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) =
        data((q ## read_data(pm, dt1)(addr1))(s))

End Plain_Mem_Rewrites6


Plain_Mem_Rewrites7 [State, Data1, Data2 : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites6
```

```
pm : VAR Plain_Memory[State]
s : VAR State
expr : VAR [State -> ExprResult[State, Data1]]
expr1 : VAR [State -> ExprResult[State, Data2]]

dt1 : Var (interpreted_data_type?[Data1])
dt2 : Var (interpreted_data_type?[Data2])
addr1, addr2 : Var Address
data1 : Var Data1
data2 : Var Data2

% proof status :-)
pm_q_prop_read_write_other_q_stmt : Lemma %(P: 6)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
  Implies
    pm_q_prop_read(pm, dt1, addr1)((q ## write_data(pm, dt2)(addr2, data2))(s))

% proof status :-)
% plain_memory_read_write_other_single_data_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_write_other_single_data_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_write_other_single_data : Lemma %(P: 4)
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) And
    pm`states(s)
  Implies
    data((write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
      data((read_data(pm, dt1)(addr1))(s))

% proof status :-)
pm_q_prop_read_read_q_stmt : Lemma %(P: 6)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s)) And
    pm_q_prop_read(pm, dt2, addr2)(q(s))
  Implies
    pm_q_prop_read(pm, dt2, addr2)((q ## read_data(pm, dt1)(addr1))(s))

% proof status :-)
```

```
pm_q_prop_read_ok_result_q_stmt : Lemma % (P: 4)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    pm_q_prop_read(pm, dt2, addr2)(q(s))
  Implies
    pm_q_prop_read(pm, dt2, addr2)((q ## ok_result[State, Data1](data1))(s))

% proof status :-)
% plain_memory_read_write_other_q_data_stmt_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_write_other_q_data_stmt_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_write_other_q_data_stmt : Lemma %(P: 6)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
  Implies
    data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
      data((q ## read_data(pm, dt1)(addr1))(s))

% proof status :-)
% plain_memory_read_read_q_data_stmt_TCC1 : TCC Obligation

% proof status :-)
% plain_memory_read_read_q_data_stmt_TCC2 : TCC Obligation

% proof status :-)
plain_memory_read_read_q_data_stmt : Lemma %(P: 6)
  Forall (q : [State -> StmtResult[State]]) :
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
    in_blessed_memory?(dt2, addr2, union(pm`ro_addr, pm`rw_addr)) And
    pm_q_prop_read(pm, dt2, addr2)(q(s)) And
    pm_q_prop_read(pm, dt1, addr1)(q(s))
  Implies
    data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) =
      data((q ## read_data(pm, dt1)(addr1))(s))

% proof status :-)
pm_q_prop_read_e2s : Lemma
  pm_q_prop_read(pm, dt1, addr1)(expr1(s)) Implies
    pm_q_prop_read(pm, dt1, addr1)(e2s(expr1)(s))

stmt : VAR [State -> StmtResult[State]]

% proof status :-)
pm_q_prop_read_stmt_e2s : Lemma
```

```
    pm_q_prop_read(pm, dt1, addr1)(stmt(s)) Implies
      pm_q_prop_read(pm, dt1, addr1)((stmt ## e2s(expr1))(s)) =
        pm_q_prop_read(pm, dt1, addr1)((stmt ## expr1)(s))

End Plain_Mem_Rewrites7


Plain_Mem_Rewrites8 [State, Data1, Data2, Data_q : Type] : Theory
Begin

  Importing Plain_Mem_Rewrites7

  pm : VAR Plain_Memory[State]
  s : VAR State

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  addr1, addr2 : Var Address
  data1 : Var Data1
  data2 : Var Data2


  % proof status :-)
  % plain_memory_read_write_other_q_data_expr_TCC1 : TCC Obligation

  % proof status :-)
  % plain_memory_read_write_other_q_data_expr_TCC2 : TCC Obligation

  % proof status :-)
  plain_memory_read_write_other_q_data_expr : Lemma %(P: 6)
    Forall (q : [State -> ExprResult[State, Data_q]]) :
      plain_memory?(pm) And
      in_blessed_memory?(dt1, addr1, union(pm`ro_addr, pm`rw_addr)) And
      in_blessed_memory?(dt2, addr2, pm`rw_addr) And
      blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
      pm_q_prop_read(pm, dt1, addr1)(q(s))
    Implies
      data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
        data((q ## read_data(pm, dt1)(addr1))(s))


End Plain_Mem_Rewrites8


Plain_Mem_Rewrites : Theory
Begin

  Importing Plain_Mem_Rewrites8

End Plain_Mem_Rewrites
```

```
Rewrite_Test[(Importing IA32) max : Memory_Address,
                    State, Data1, Data2 : Type] : Theory
Begin

  Importing Plain_Memory[State]
  Importing Plain_Mem_Rewrites

  pm : Var Plain_Memory[State]

  dt1 : Var (interpreted_data_type?[Data1])
  dt2 : Var (interpreted_data_type?[Data2])
  s : Var State
  addr1, addr2 : Var Address
  data1 : Var Data1
  data2 : Var Data2


  % Test for Plain Mem Rewrites
  %-------------------------------

  % proof status :-)
  rewrite_test : Lemma
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, pm`rw_addr) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
    pm`states(s)
  Implies
    OK?(
     (write_data(pm, dt1)(addr1, data1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      read_data(pm, dt1)(addr1)
     )(s))

  % proof status :-)
  % rewrite_test_data_TCC1 : TCC Obligation

  % proof status :-)
  rewrite_test_data : Lemma
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, pm`rw_addr) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
    pm`states(s)
  Implies
    data(
     (write_data(pm, dt1)(addr1, data1) ##
```

```
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      read_data(pm, dt1)(addr1)
     )(s)) = data1

% proof status :-)
% rewrite_test_data_long_TCC1 : TCC Obligation

% proof status :-)
rewrite_test_data_long : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) And
  in_blessed_memory?(dt2, addr2, pm`rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
  pm`states(s)
Implies
  data(
     (write_data(pm, dt1)(addr1, data1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
```

```
      read_data(pm, dt2)(addr2) ##
      read_data(pm, dt1)(addr1)
    )(s)) = data1

% proof status :-)
rewrite_test_data_eval_if_ok_ok : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) And
  in_blessed_memory?(dt2, addr2, pm`rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
  pm`states(s)
Implies
  OK?(
   (write_data(pm, dt1)(addr1, data1) ##
    write_data(pm, dt2)(addr2, data2) ##
    read_data(pm, dt1)(addr1) ##
    write_data(pm, dt2)(addr2, data2) ##
    read_data(pm, dt2)(addr2) ##
      Lambda(d : Data2) :
        read_data(pm, dt1)(addr1))(s))


% proof status :-)
% rewrite_test_data_eval_if_ok_TCC1 : TCC Obligation

% proof status :-)
rewrite_test_data_eval_if_ok : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) And
  in_blessed_memory?(dt2, addr2, pm`rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
  pm`states(s)
Implies
  data(
   (write_data(pm, dt1)(addr1, data1) ##
    write_data(pm, dt2)(addr2, data2) ##
    read_data(pm, dt1)(addr1) ##
    write_data(pm, dt2)(addr2, data2) ##
    read_data(pm, dt2)(addr2) ##
      Lambda(d : Data2) :
        read_data(pm, dt1)(addr1)
   )(s)) = data1

% proof status :-)
rewrite_test_data_eval_if_ok_ok_result_ok : Lemma
  plain_memory?(pm) And
  in_blessed_memory?(dt1, addr1, pm`rw_addr) And
  in_blessed_memory?(dt2, addr2, pm`rw_addr) And
  blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
  blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
```

```
    pm`states(s)
  Implies
    OK?[State, Data2](
     (write_data(pm, dt1)(addr1, data1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
        Lambda(d : Data2) :
          ok_result(d)
     )(s))

  % proof status :-)
  % rewrite_test_data_eval_if_ok_ok_result_TCC1 : TCC Obligation

  % proof status :-)
  rewrite_test_data_eval_if_ok_ok_result : Lemma
    plain_memory?(pm) And
    in_blessed_memory?(dt1, addr1, pm`rw_addr) And
    in_blessed_memory?(dt2, addr2, pm`rw_addr) And
    blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) And
    blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) And
    pm`states(s)
  Implies
    data(
     (write_data(pm, dt1)(addr1, data1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt1)(addr1) ##
      write_data(pm, dt2)(addr2, data2) ##
      read_data(pm, dt2)(addr2) ##
        Lambda(d : Data2) :
          ok_result(d)
     )(s)) = data2

End Rewrite_Test

% Rewrites: Plain Memory
% (auto-rewrite
% "plain_memory_write_ok_single"
% "plain_memory_write_ok_q_expr"
% "plain_memory_write_ok_q_stmt"
% "in_blessed_memory_rw_ro"
% "pm_q_prop_ok"
% "pm_q_prop_single_read"
% "pm_q_prop_single_write"
% "pm_q_prop_read_write_single"
% "pm_q_prop_read_write_other_single"
% "pm_q_prop_read_read_single"
% "plain_memory_read_write_other_single_data"
% "plain_memory_read_read"
% "plain_memory_read_write_res"
% "pm_q_prop_read_ok_expr"
```

% "pm_q_prop_read_ok_stmt"
% "pm_q_prop_read_pm_q_prop_expr"
% "pm_q_prop_read_pm_q_prop_stmt"
% "pm_q_prop_read_write_q_expr"
% "pm_q_prop_read_write_q_stmt"
% "plain_memory_read_write_q_data_expr"
% "plain_memory_read_write_q_data_stmt"
% "pm_q_prop_read_write_other_q_expr"
% "pm_q_prop_read_write_other_q_stmt"
% "pm_q_prop_read_read_q_expr"
% "pm_q_prop_read_read_q_stmt"
% "plain_memory_read_write_other_q_data_expr"
% "plain_memory_read_write_other_q_data_stmt"
% "plain_memory_read_read_q_data_expr"
% "plain_memory_read_read_q_data_stmt"
% "pm_q_prop_ok_result_single"
% "pm_q_prop_ok_result_q_expr"
% "pm_q_prop_ok_result_q_stmt"
% "pm_q_prop_read_ok_result_single"
% "pm_q_prop_read_ok_result_q_expr"
% "pm_q_prop_read_ok_result_q_stmt"
% "pm_q_prop_e2s"
% "pm_q_prop_stmt_e2s"
% "pm_q_prop_read_e2s"
% "pm_q_prop_read_stmt_e2s"
%)

B.24 ptab-sync-master-defs.pvs

```
reinterpret_cast_thy [State, Data1, Data2 : Type] : Theory
Begin

  Importing Cpp_Verification

  dt1 : VAR (interpreted_data_type?[Data1])
  dt2 : VAR (interpreted_data_type?[Data2])

  reinterpret_cast(dt1, dt2)(ex : ET[State, Data2]) : ET[State, Data1] =
  (
    ex ## lambda (d2 : Data2) :
    allocate_stack(uidt(dt2)) ## lambda (a : Address) :
    write_data(pm, dt2)(a, d2) ##
    read_data(pm, dt1)(a) ## lambda (d1 : Data1) :
    deallocate_stack(uidt(dt2), a) ##
    ok_result[State, Data1](d1)
  )

End reinterpret_cast_thy


ptab_types : Theory
```

```
Begin

  Paging : Type
  Buddy : Type
  Ptab : Type

  void : Type

End ptab_types


ptab_sync_master_defs_state_transformers [State : Type] : Theory
Begin

  Importing Cpp_Verification, ptab_types

  static_cast_nocv_unsigned_long_long_reference_to_nocv_unsigned_long_int
                          (ex : ET[State, (range(ulonglong))]) : ET[State, (range(ulong))] =
  (
    ex ## lambda (t : (range(ulonglong))) :
    If true Then % (TODO) FIXME: dt_pred(ulonglong)(t) Then
      ok_result(t)
    Else
      fatal_result
    Endif
  )

  static_cast_nocv_void_nocv_pointer_to_nocv_Ptab_nocv_pointer
                          (ex : ET[State, (range(pointer(void)))]) : ET[State, (range(pointer(class("Ptab", false))))]

  int_conv_nocv_unsigned_long_int_reference_to_const_unsigned_int
                          (ex : ET[State, (range(ulong))]) : ET[State, (range(uint))]

  int_conv_nocv_unsigned_long_int_to_nocv_unsigned_int
                          (ex : ET[State, (range(ulong))]) : ET[State, (range(uint))]

End ptab_sync_master_defs_state_transformers


ptab_sync_master_defs : Theory
Begin

  Importing ptab_sync_master_defs_state_transformers, reinterpret_cast_thy

  % To do:
  %========
  % - generate statically initialised variables and constants
  % - generate offsets

  % offsets
% MV: >> fix pointer
```

```
  Ofs_Paging_type : Type = [#
    val : nat
  #]

  offsets_Paging : Ofs_Paging_type

  Ofs_Buddy_type : Type = [#
    l_base : nat,
    r_base : nat,
    p_base : nat
  #]

  offsets_Buddy : Ofs_Buddy_type

  % Enum
  ATTR_PRESENT : (range(ulonglong)) = shift_bits_right(1, 0)
  ATTR_NOEXEC : (range(ulonglong)) = shift_bits_right(1, 63)
  ATTR_ALL : (range(ulonglong)) = 0 %bitor(ATTR_NOEXEC, (shift_bits_right(1, 12) - 1))

  % static members; static const initialisation is missing
  levels : Address
  bits_per_level : Address

  allocator : Address

  Ptab : Cpp_Type

End ptab_sync_master_defs
```

B.25 ptab-sync-master.pvs

```
ptab_sync_master[State : Type] : THEORY
BEGIN

  IMPORTING Cpp_Verification
  IMPORTING ptab_sync_master_defs

  % Function semantics for addr
  % File "nova/include/paging.h", line 69, character 9
  addr(receiver : Address)(ret_addr : Address) : ST[State] =
    return(pm, dt_ulonglong, ret_addr)(bitand(ulonglong)(member(id(receiver),
                                                                offsets_Paging`val),
                                                               unary_bitnot(int)(id(ATTR_ALL))))

  call_addr(receiver : ET[State, Semantics_reference[Paging]]) : ET[State,
              Semantics_ulonglong] =
    with_new_returnvar(pm, dt_ulonglong)(lambda (ret_addr : Address) :
      receiver ##
        (lambda (receiver_addr : Address):
        addr(receiver_addr)(ret_addr)))
```

```
% Function semantics for operator=
% File "nova/include/paging.h", line 16, character 1
operator_assign_Paging(receiver : Address, other : Address)(ret_addr : Address)
  : ST[State] =
  e2s(assign(pm, dt_ulonglong)(member(id(receiver), offsets_Paging`val),
                               member(id(other), offsets_Paging`val))) ##
  return(pm, dt_reference[Paging], ret_addr)(id(receiver))

call_operator_assign_Paging(receiver : ET[State,
                                          Semantics_reference[Paging]],
                                          other : ET[State, Semantics_reference[Paging]])
                                          : ET[State, Semantics_reference[Paging]] =
  with_new_returnvar(pm, dt_reference[Paging])(lambda (ret_addr : Address) :
    other ##
      (lambda (other_addr : Address):
      receiver ##
        (lambda (receiver_addr : Address):
        operator_assign_Paging(receiver_addr, other_addr)(ret_addr))))


% Function semantics for operator=
% File "nova/include/ptab.h", line 20, character 1
operator_assign_Ptab(receiver : Address, other : Address)(ret_addr : Address)
  : ST[State] =
  e2s(call_operator_assign(id(receiver), id(other))) ##
  return(pm, dt_reference[Ptab], ret_addr)(id(receiver))

call_operator_assign_Ptab(receiver : ET[State, Semantics_reference[Ptab]],
                                     other : ET[State, Semantics_reference[Ptab]]) :
                                     ET[State, Semantics_reference[Ptab]] =
  with_new_returnvar(pm, dt_reference[Ptab])(lambda (ret_addr : Address) :
    other ##
      (lambda (other_addr : Address):
      receiver ##
        (lambda (receiver_addr : Address):
        operator_assign_Ptab(receiver_addr, other_addr)(ret_addr))))


% Function semantics for present
% File "nova/include/paging.h", line 81, character 9
present(receiver : Address)(ret_addr : Address) : ST[State] =
  return(pm, dt_ulonglong, ret_addr)(bitand(ulonglong)(member(id(receiver),
                                                              offsets_Paging`val),
                                                             id(ATTR_PRESENT)))

call_present(receiver : ET[State, Semantics_reference[Paging]]) : ET[State,
               Semantics_bool] =
  with_new_returnvar(pm, dt_bool)(lambda (ret_addr : Address) :
    receiver ##
      (lambda (receiver_addr : Address):
      present(receiver_addr)(ret_addr)))
```

```
% Function semantics for phys_to_virt
% File "nova/include/buddy.h", line 74, character 9
phys_to_virt(receiver : Address, phys : Address)(ret_addr : Address) :
  ST[State] =
  return(pm, dt_ulong, ret_addr)(plus(ulong)(minus(ulong)(id(phys),
                                                           member(id(receiver),
                                                           offsets_Buddy`p_base)),
                                                  member(id(receiver),
                                                    offsets_Buddy`l_base)))

call_phys_to_virt(receiver : ET[State, Semantics_reference[Buddy]],
                  phys : ET[State, Semantics_ulong]) : ET[State,
                  Semantics_ulong] =
  with_new_returnvar(pm, dt_ulong)(lambda (ret_addr : Address) :
    with_new_stackvar(dt_ulong)(lambda (phys_addr : Address):
      e2s(assign(pm, dt_ulong)(id(phys_addr), phys)) ##
      receiver ##
        (lambda (receiver_addr : Address):
        phys_to_virt(receiver_addr, phys_addr)(ret_addr))))


% Function semantics for phys_to_ptr
% File "nova/include/buddy.h", line 97, character 9
phys_to_ptr(phys : Address)(ret_addr : Address) : ST[State] =
  return(pm, dt_pointer[void], ret_addr)(reinterpret_cast(dt_pointer[void],
                                                           dt_ulong)
  (call_phys_to_virt(id(allocator),
                     static_cast_nocv_unsigned_long_long_reference_to_nocv_unsigned_long_int
                     (id(phys)))))

call_phys_to_ptr(phys : ET[State, Semantics_ulonglong]) : ET[State,
                 Semantics_pointer[void]] =
  with_new_returnvar(pm, dt_pointer[void])(lambda (ret_addr : Address) :
    with_new_stackvar(dt_ulonglong)(lambda (phys_addr : Address):
      e2s(assign(pm, dt_ulonglong)(id(phys_addr), phys)) ##
      phys_to_ptr(phys_addr)(ret_addr)))


% Function semantics for master
% File "nova/include/ptab.h", line 36, character 9
master(ret_addr : Address) : ST[State] =
  with_new_stackvar(lambda (master_l : Address):
    e2s(skip) ##
    return(pm, dt_Address, ret_addr)(id(master_l)))

call_master : ET[State, Semantics_pointer[Ptab]] =
  with_new_returnvar(pm, dt_pointer[Ptab])(lambda (ret_addr : Address) :
    master)


% Function semantics for sync_master
```

```
% File "nova/ptab.cpp", line 203, character 1
sync_master(receiver : Address, virt : Address)(ret_addr : Address) :
  ST[State] =
  with_new_stackvar(lambda (lev : Address):
    with_new_stackvar(lambda (pte : Address):
      with_new_stackvar(lambda (mst : Address):
        e2s(assign(pm, dt_uint)(id(lev), l2r(pm, dt_uint)(id(levels)))) ##
        e2s(skip ## skip) ##
        for(e2s(comma(reference[pointer[Ptab]])(assign(pm, dt_pointer[Ptab])
                                                (id(pte), id(receiver)),
                                                assign(pm, dt_pointer[Ptab])
                                                (id(mst), call_master))),
          n2b(literal(true)),
          comma(reference[pointer[Ptab]])(assign(pm, dt_pointer[Ptab])
                                          (id(pte),
                                            static_cast_nocv_void_nocv_pointer_to_nocv_Ptab_nocv_pointer
                                            (call_phys_to_ptr(
                                               call_addr(id(pte))))),
                                          assign(pm, dt_pointer[Ptab])
                                          (id(mst),
                                            static_cast_nocv_void_nocv_pointer_to_nocv_Ptab_nocv_pointer
                                            (call_phys_to_ptr(
                                               call_addr(id(mst)))))),
          with_new_stackvar(lambda (shift : Address):
            with_new_stackvar(lambda (slot : Address):
              with_new_stackvar(lambda (size : Address):
                e2s(assign(pm, dt_uint)(id(shift),
                                        plus(uint)(times(uint)
                                                   (predec(pm, reference[uint])(id(lev)),
                                                     id(bits_per_level)),
                                                   literal[State, Semantics_int](12)))) ##
                e2s(assign(pm, dt_uint)(id(slot),
                                        int_conv_nocv_unsigned_long_int_to_nocv_unsigned_int(
                                        bitand(ulong)(shr(ulong)
                                                      (id(virt),
                                                       id(shift)),
                                                      minus(ulong)
                                                      (shl(ulong)
                                                       (literal[State, Semantics_ulong](1),
                                                       id(bits_per_level)),
                                                       literal[State, Semantics_int](1)))))) ##
                e2s(assign(pm, dt_uint)(id(size),
                                        int_conv_nocv_unsigned_long_int_to_nocv_unsigned_int(
                                        shl(ulong)(literal[State, Semantics_ulong](1),
                                                   id(shift))))) ##
                e2s(assign_plus(pm, pointer[Ptab])(id(mst),
                                                   l2r(pm, dt(int))
                                                   (id(slot)))) ##
                e2s(assign_plus(pm, pointer[Ptab])(id(pte),
                                                   l2r(pm, dt(int))
                                                   (id(slot)))) ##
                if_else(call_present(id(mst)),
```

```
                    if_else(eq(bool)(id(slot),
                                     bitand(uint)(shr(uint)(literal[State, Semantics_uint](−807403520),
                                                            id(shift)),
                                                  minus(ulong)
                                                  (shl(ulong)
                                                    (literal[State, Semantics_ulong](1),
                                                      id(bits_per_level)),
                                                    literal[State, Semantics_int](1)))),
                      skip ##
                      continue,
                      skip) ##
                    e2s(call_operator_assign(id(pte), id(mst))) ##
                    skip,
                    skip) ##
                  return(pm, dt_reference[uint], ret_addr)(id(size))))))))) ##
  return_void

call_sync_master(receiver : ET[State, Semantics_reference[Ptab]],
                 virt : ET[State, Semantics_ulong]) : ET[State,
                 Semantics_uint] =
  with_new_returnvar(pm, dt_uint)(lambda (ret_addr : Address) :
    with_new_stackvar(dt_ulong)(lambda (virt_addr : Address):
      e2s(assign(pm, dt_ulong)(id(virt_addr), virt)) ##
      receiver ##
        (lambda (receiver_addr : Address):
        sync_master(receiver_addr, virt_addr)(ret_addr))))


END ptab_sync_master
```

B.26 random_device.pvs

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Random Device
%
% base : base address of the device
% [base + rnd_seed] : Word (32 bit) − seed value % 2^32
% [base + mem_cnt] : Word − memory access count % 2^32 ;
% writing to mem_cnt clears the counter to 0
% [base + rnd_val] : Word − random value based on choose(seed x cnt x N);
% the value is random as long as states are
% initialised with differing seed values
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Random_Device[Physical_memory : Type,
              (Importing Plain_Memory)
              pm : Plain_Memory[Physical_memory]] : Theory
Begin

  Assuming
```

```
  Importing Cpp_Types

  dt_uint_size : Assumption
    0 < size(uidt(dt_uint)) And 3 * size(uidt(dt_uint)) <= max_linear_offset

EndAssuming

% Device State
%-----------------
Random_device_state : Type = [#
  seed : nat,
  memory_access_count : nat,
  counter_offset : {n : nat | n <= memory_access_count}
#]

% proof status :-)
% base_TCC1 : TCC Obligation

size : nat = 3 * size(uidt(dt_uint))
base : ({m : Memory_Address_4G | m + size <= max_linear})

% proof status :-)
% rnd_seed_TCC1 : TCC Obligation

% proof status :-)
% mem_cnt_TCC1 : TCC Obligation

% proof status :-)
% rnd_val_TCC1 : TCC Obligation

rnd_seed : Memory_Address_4G = base + 0 * size(uidt(dt_uint))
mem_cnt : Memory_Address_4G = base + 1 * size(uidt(dt_uint))
rnd_val : Memory_Address_4G = base + 2 * size(uidt(dt_uint))

Importing Device_Memory[Physical_memory, pm, Random_device_state]

Random_device_memory : Type = Device_memory

s : Var Random_device_memory

% Accessor functions
%--------------------
seed(s) : nat = s`expansion`seed
access_count(s) : nat = s`expansion`memory_access_count
counter_offset(s) : nat = s`expansion`counter_offset

% proof status :-)
% increase_access_count_TCC1 : TCC Obligation

increase_access_count(s)(n : nat) : Random_device_memory =
  s With [(expansion) := s`expansion With [(memory_access_count) := access_count(s) + n]]
```

```
  % proof status :-)
  % clear_counter_TCC1 : TCC Obligation

  clear_counter(s) : Random_device_memory =
    s WITH [(expansion) := s`expansion WITH [(counter_offset) := access_count(s)]]

  % Side Effects
  %-----------------
  access_count(a : Address, bl : list[Byte], cp : bool)(s) :
                                      ExprResult[Random_device_memory, list[Byte]] =
    OK(increase_access_count(s)(length(bl)), bl)

  unaligned_access(a : Address, bl : list[Byte], cp : bool) :
        [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
    IF null?(bl) OR disjoint?(address_block(a, length(bl)),
                                  address_block(base, size)) OR
        % check aligned access
        (NOT cp AND length(bl) = size(uidt(dt_uint)) AND
        (a = rnd_seed OR a = mem_cnt OR a = rnd_val)) THEN
      ok_result(bl)
    ELSE
      fatal_result
    ENDIF

  % proof status :-)
  % random_TCC1 : TCC Obligation

  random(seed : nat, count : nat) :
    {l : list[Byte] | length(l) = size(uidt(dt_uint))}

  random(s : Random_device_memory) : list[Byte] =
    random(seed(s), access_count(s))

  read_rnd_val(a : Address, bl : list[Byte], cp : bool) :
        [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
    IF cons?(bl) AND a = rnd_val THEN
      LAMBDA (s : Random_device_memory) : ok_result(random(s))(s)
    ELSE
      ok_result(bl)
    ENDIF

  % proof status :-)
  % read_mem_cnt_TCC1 : TCC Obligation

  % proof status :-)
  % read_mem_cnt_TCC2 : TCC Obligation

  % proof status :-)
  % read_mem_cnt_TCC3 : TCC Obligation

  read_mem_cnt(a : Address, bl : list[Byte], cp : bool) :
        [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
```

```
  If cons?(bl) And a = mem_cnt Then
    Lambda (s : Random_device_memory) :
      ok_result(dt_uint`to_byte(
        rem(max_value(uint))(access_count(s) - counter_offset(s)), mem_cnt))(s)
  Else
    ok_result(bl)
  Endif

% proof status :-)
% read_seed_TCC1 : TCC Obligation

% proof status :-)
% read_seed_TCC2 : TCC Obligation

% proof status :-)
% read_seed_TCC3 : TCC Obligation

read_seed(a : Address, bl : list[Byte], cp : bool) :
      [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
  If cons?(bl) And a = rnd_seed Then
    Lambda (s : Random_device_memory) : ok_result(
      dt_uint`to_byte(rem(max_value(uint))(seed(s)), rnd_seed))(s)
  Else
    ok_result(bl)
  Endif

write_rnd_dev(a : Address, bl : list[Byte], cp : bool) :
      [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
  If cons?(bl) And (a = rnd_val Or a = rnd_seed) Then
    fatal_result
  Else
    ok_result(bl)
  Endif

reset_count(a : Address, bl : list[Byte], cp : bool)(s) :
                                  ExprResult[Random_device_memory, list[Byte]] =
  If cons?(bl) And a = mem_cnt Then
    OK(clear_counter(s), bl)
  Else
    ok_result(bl)(s)
  Endif

% read side effects
rnd_dev_read_side_effect(a : Address, bl : list[Byte], cp : bool) :
      [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
(
  unaligned_access(a, bl, cp) ## Lambda (bl : list[Byte]) :
  access_count(a, bl, cp) ## Lambda (bl : list[Byte]) :
  read_rnd_val(a, bl, cp) ## Lambda (bl : list[Byte]) :
  read_mem_cnt(a, bl, cp) ## Lambda (bl : list[Byte]) :
  read_seed(a, bl, cp)
)
```

```
  rnd_dev_write_side_effect(a : Address, bl : list[Byte], cp : bool) :
        [Random_device_memory -> ExprResult[Random_device_memory, list[Byte]]] =
  (
    unaligned_access(a, bl, cp) ## Lambda (bl : list[Byte]) :
    access_count(a, bl, cp) ## Lambda (bl : list[Byte]) :
    write_rnd_dev(a, bl, cp) ## Lambda (bl : list[Byte]) :
    reset_count(a, bl, cp)
  )

  % Plain Memory
  %==============

  % Memory Struct
  %----------------
  random_device_pm_mem : Memory_struct[Random_device_memory] =
    device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect)

  % Random_Device PM Struct
  %------------------------

  random_device_pm : Plain_Memory[Random_device_memory] = (#
    mem := random_device_pm_mem,
    states := em_lift(pm`states),
    ro_addr := difference(pm`ro_addr, address_block(base, size)),
    rw_addr := difference(pm`rw_addr, address_block(base, size)),
    other_actions := em_lift(pm`other_actions)
  #)

End Random_Device

Challenge_Random_Device[Physical_memory : Type,
                        (Importing Plain_Memory)
                        pm_phy : Plain_Memory[Physical_memory]] : Theory
Begin

  Assuming

    Importing Cpp_uint, Extended_Real

    dt_uint_size : Assumption
      0 < size(uidt(dt_uint)) And 3 * size(uidt(dt_uint)) <= max_linear_offset

  EndAssuming

  % proof status :-)
  % IMP_Random_Device_TCC1 : TCC Obligation

  Importing Random_Device[Physical_memory, pm_phy],
            Challenge_Device_Memory[Physical_memory, pm_phy, Random_device_state]

  % Side Effect Conditions
```

```
%-----------------------

pm : Plain_Memory[Random_device_memory] = random_device_pm

% proof status :-)
in_blessed_memory_disjoint_device : Lemma
  Forall (a : Address, l : nat) :
    subset?(address_block(a, l), union(pm`ro_addr, pm`rw_addr))
      Implies
    disjoint?(address_block(a, l), address_block(base, size))

% proof status :-)
in_blessed_memory_not_device_address : Lemma
  Forall (a : Address, bl : list[Byte]) :
    subset?(address_block(a, length(bl)), union(pm`ro_addr, pm`rw_addr)) And cons?(bl)
      Implies
    a /= rnd_seed And a/= mem_cnt And a /= rnd_val

% proof status :-)
random_side_effects_unchanged_invariant : Lemma
  unchanged_memory_invariant?(pm`mem, pm`states, union(
      drse_super_transformers(union(pm`ro_addr, pm`rw_addr),
                              rnd_dev_read_side_effect),
      dwse_super_transformers(pm`rw_addr, rnd_dev_write_side_effect)),
                              union(pm`ro_addr, pm`rw_addr))

% proof status :-)
random_side_effects_transformers_ok : Lemma
  transformers_ok?(pm`states, union(
    drse_super_transformers(union(pm`ro_addr, pm`rw_addr),
                            rnd_dev_read_side_effect),
    dwse_super_transformers(pm`rw_addr, rnd_dev_write_side_effect)))

% proof status :-)
random_side_effects_side_effect_content_unchanged_read : Lemma
  side_effect_content_unchanged(union(pm`ro_addr, pm`rw_addr), pm`states,
                                rnd_dev_read_side_effect)

% proof status :-)
random_side_effects_side_effect_content_unchanged_write : Lemma
  side_effect_content_unchanged(pm`rw_addr, pm`states,
                                rnd_dev_write_side_effect)

% Plain Memory
%==============
% proof status :-)
random_device_plain_memory : Lemma
  plain_memory?(pm_phy) Implies
    plain_memory?(pm)

% Fail on Unaligned Accesses
%============================
```

```
  % proof status :-)
  unaligned_fails_write : Lemma
    Forall (s : Random_device_memory, a : Address, bl : (cons?[Byte])) :
      Not disjoint?(address_block(a, length(bl)), address_block(base, size)) And
        Not (length(bl) = size(uidt(dt_uint)) And (a = rnd_seed Or a = mem_cnt Or a = rnd_val)) And
          % underlying write side effect is executed first ; require it to be ok
          OK?(memory_write_side_effect(pm_phy`mem)(a, bl, false)(s`state)) And
          data(memory_write_side_effect(pm_phy`mem)(a, bl, false)(s`state)) = bl
    Implies
      Fatal?(memory_write_list(pm`mem)(a, bl)(s))

  % proof status :-)
  unaligned_fails_read : Lemma
    Forall (s : Random_device_memory, a : Address, bl : (cons?[Byte])) :
      Not disjoint?(address_block(a, length(bl)), address_block(base, size)) And
        Not (length(bl) = size(uidt(dt_uint)) And (a = rnd_seed Or a = mem_cnt Or a = rnd_val)) And
          % underlying read list and read side effect is executed first ; require it to be ok
          OK?(memory_read_list_nse(pm`mem)(a, length(bl))(s)) And
          (Let ures = (memory_read_list_nse(pm`mem)(a, length(bl)) ##
                       LAMBDA (bl1: list[Byte]): em_lift[Physical_memory,
                                                         Random_device_state[Physical_memory, pm_phy], list[Byte]
                       (memory_read_side_effect(pm_phy`mem)
                            (a, bl1, false)))(s) In
             OK?(ures) And length(data(ures)) = length(bl))
    Implies
      Fatal?(memory_read_list(pm`mem)(a, length(bl))(s))

End Challenge_Random_Device
```

B.27 result.pvs

```
% $Id: result.pvs,v 1.32.10.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Hendrik Tews
% Author: Marcus Voelp
% Author: Tjark Weber, (c) 2007 Radboud University
%
% Result types for C++ expressions and statements.

% TODO: The Pagefault/Exception related definitions need cleanup when it is
% clear what exactly we want to model (and at which level). Perhaps
% they could be moved to a different file also.


Memory_access: DATATYPE
BEGIN
    Read : Read?
    Write : Write?
    Execute : Execute?
END Memory_access
```

```
% This differes from CPU privilege levels in that there are only two.
% We assume that CPL 1 and CPL 2 are never ever used.
Memory_privilege : DATATYPE
BEGIN
    Supervisor : Supervisor?
    User : User?
END Memory_privilege


Memory_access_util : Theory
% utility function for Memory_access
Begin
  Importing Memory_access

  RX_Memory_access : Type = {p : pred[Memory_access] | subset?(add(Execute, singleton(Read)), p)}
  R_Memory_access : Type = {p : pred[Memory_access] | member(Read, p)}

  memory_access_to_bools(ar : PRED[Memory_access]) : [bool, bool] =
    (member(Write, ar), member(Execute, ar))

  bools_to_memory_access(write : bool, fetch : bool) : PRED[Memory_access] =
    If write Then
      If fetch Then
        add(Execute, add(Write, singleton(Read)))
      Else
        add(Write, singleton(Read))
      Endif
    Else
      If fetch Then
        add(Execute, singleton(Read))
      Else
        singleton(Read)
      Endif
    Endif

End Memory_access_util


Memory_privilege_util : Theory
% utility functions for Memory_privilege
Begin
  Importing Memory_privilege

  memory_privilege_to_bool(ar : Memory_privilege) : bool =
    Cases ar of
      Supervisor : false,
      User : true
    EndCases

  bool_to_memory_privilege(b : bool) : Memory_privilege =
    IF b Then User Else Supervisor Endif
```

```
  % proof status :-)
  bool_to_memory_privilege_iso : Lemma Forall(ar : Memory_privilege) :
    bool_to_memory_privilege(memory_privilege_to_bool(ar)) = ar

End Memory_privilege_util


Error_Code_Types : Theory
Begin
  Importing Memory_privilege, Memory_access_util

  PF_error_code_type : Type = [#
    reserved_bit_violation : bool, % page fault not modeled
    privilege : Memory_privilege,
    access : R_Memory_access,
    present : bool
  #]

End Error_Code_Types


Exception_type : Datatype
Begin

  Importing Error_Code_Types

  % Exceptions which may possibly occur in the kernel and which are
  % smoothly handled. If a not listed exceptions occurs in kernel code
  % this is a fatal error. The following lists when these exceptions
  % occur within the kernel:

  % - the kernel copies from / to user memory
  % - sync with master page tables
  Page_fault(ec : PF_error_code_type) : Page_fault?

  % - lazy segment loads
  General_protection : General_protection? %

End Exception_type


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% ExprResult: the result type for C++ expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ExprResult[State, Data : Type] : DATATYPE
BEGIN

%% IMPORTING Pagefault_Flag
  IMPORTING Exception_type
```

```
  % result OK with successor state and result data
  OK(state: State, data: Data) : OK?

  % not (yet) delivered exception or interrupt
  % Before the exception handler can be invoked the exception must be
  % delivered. At delivery the IDT is traversed and the conditions for
  % double faults need to be checked and the stack frame must be
  % constructed.
  Exception(
    ex_type : Exception_type,
    state : State % state during which exception occured
  ) : Exception?

%% % not (yet) handled page fault
%% Page_fault(
%% pfa : Address,
%% page_fault_flags : Pagefault_flag) : Page_fault?
%%
%% % TLB inconsistent with current page directory
%% TLB_fault : TLB_fault?
%%
%% % general protection fault (in case somebody wants to handle those)
%% % XXX should all exceptions get a result?
%% GP_fault : GP_fault?
%%
%% % return from interrupt
%% % Attention: this result must be caught immediately. It must not
%% % be passed through other state transformers!
%% Iret(next_state : State) : Iret?
%%

  % worst case crash (undefined behavior)
  Fatal : Fatal?

  % a non-terminating program statement has been encountered
  Hang : Hang?

END ExprResult


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% StmtResult: the result type for C++ statements
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

StmtResult[State : Type] : DATATYPE
BEGIN
  Importing Exception_type

  % result OK with successor state
```

```
  % Note that this, unlike the "ExprResult.OK" constructor, does not have a
  % "data" argument. (If it had, we would have to invent a term of type "Data"
  % at some places in the C++ program, e.g. at the beginning of a function, or
  % after catching a break/continue.)
  OK(state : State) : OK?

  % "break" has been executed
  Break(state : State) : Break?

  % "continue" has been executed
  Continue(state : State) : Continue?

  % "return" has been executed
  % We model return expr as expr ## lambda (res : Data) : write(ret_addr, data)
  % where ret_addr can be an unused region of the stack frame suitable
  % to hold the returned type or a register of suitable size.
  % This model includes returning classes by allocating them in a
  % caller allocated memory and returning integral types in the eax register as
  % specified in the Unix System V ABI for Intel386 Processors.
  Return(state : State) : Return?

  % "goto label" has been executed
  % The formalization of goto statements, although possible, would introduce
  % two complications:
  % 1. C++ allows gotos into nested substatements. Thus every statement
  % potentially catches a Goto result.
  % 2. Since backwards jumps may lead to non-termination, gotos need to be
  % treated similar to loops.
  % Goto(state : State, label : string) : Goto?

  % "switch (case)" has been executed, and we're looking for a matching
  % "case"
  % Note that "case" cannot have type "Data", as that type is already reserved
  % for the result of the function in which the switch statement occurs. Thus
  % we're using the fixed type "int". Alternatively, StmtResult could take
  % another type argument "SwitchData : Type".
  Switch(state : State, case : int) : Switch?

  % a switch stamement has been executed, and we already know that there is no
  % matching "case" (hence the "default" must be taken if there is one)
  Default(state : State) : Default?

  % Not (yet) delivered exception or interrupt, see also ExprResult.
  Exception(
    ex_type : Exception_type,
    state : State % state during which exception occured
  ) : Exception?

  % worst case crash (undefined behavior)
  Fatal : Fatal?

  % a non-terminating program statement has been encountered
```

```
  Hang : Hang?

END StmtResult


% The following is a reduced super type of ExprResult and StmtResult.
% Transformer invariants and other general stuff, eg transformer_ok?,
% are formulated with SuperResult. Expression transformers and statement
% transformers can be cast into a super transformer in the obvious way:
%
% OK -> OK
% abnormalities with successor state -> Abnormal
% everything else -> Bottom
%
% See Super_Embedding below.

SuperResult[State : Type] : DATATYPE
Begin
  OK(state : State) : OK?
  Abnormal(state : State) : abnormal?
  Bottom : bottom?
End SuperResult


Super_Result_Util[State : Type] : Theory
Begin
  Importing SuperResult

  has_next_state(res : SuperResult[State]) : bool =
    Cases res OF
      OK(state) : true,
      Abnormal(state) : true,
      Bottom : false
    EndCases

End Super_Result_Util


Super_Embedding_Expr[State, Data : Type] : Theory
Begin
  Importing ExprResult, Super_Result_Util

  has_next_state(expr : ExprResult[State, Data]) : bool =
    Cases expr OF
      OK(state, date) : true,
      Exception(ex_type, state) : true,
      Fatal : false,
      Hang : false
    EndCases

  % proof status :-)
  % expr_2_super_res_TCC1 : TCC Obligation
```

```
  % proof status :-)
  % expr_2_super_res_TCC2 : TCC Obligation

  expr_2_super_res(expr : ExprResult[State, Data]) : SuperResult[State] =
    IF OK?(expr) Then OK(state(expr))
    Elsif has_next_state(expr) Then Abnormal(state(expr))
    Else Bottom
    Endif

End Super_Embedding_Expr

Super_Embedding_Stmt[State : Type] : Theory
Begin
  Importing StmtResult, Super_Result_Util

  has_next_state(stmt : StmtResult[State]) : bool =
    Cases stmt OF
      OK(state) : true,
      Break(state) : true,
      Continue(state) : true,
      Return(state) : true,
      Switch(state, case) : true,
      Default(state) : true,
      Exception(ex_type, state) : true,
      Fatal : false,
      Hang : false
    EndCases

  % proof status :-)
  % stmt_2_super_res_TCC1 : TCC Obligation

  % proof status :-)
  % stmt_2_super_res_TCC2 : TCC Obligation

  stmt_2_super_res(stmt : StmtResult[State]) : SuperResult[State] =
    IF OK?(stmt) Then OK(state(stmt))
    Elsif has_next_state(stmt) Then Abnormal(state(stmt))
    Else Bottom
    Endif

End Super_Embedding_Stmt

Super_Embedding : Theory
Begin
  Importing Super_Embedding_Stmt, Super_Embedding_Expr

End Super_Embedding
```

B.28 search-example.pvs

```
% Additional_Rewrites[State, Data : Type] : Theory
```

```
% Begin

% address_of_q
% check_bounds_q
% same_array_left_q
% same_array_right_q
% not_null_q

% End Additional_Rewrites

Search_Example[State : Type] : Theory
  Begin

  IMPORTING Cpp_Verification[State]

  % Variable and Constant Definition
  %=================================

  first : Address % pointer to first array element
  last : Address % pointer to last array element

  current : Address % pointer to current array element

  N : posnat = 2 % array size

  % proof status :-)
  % value_TCC1 : TCC Obligation

  value : (range(uint)) % value to search for

  % Preconditions
  %===============

  % proof status :-)
  % Search_Pre_TCC1 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC2 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC3 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC4 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC5 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC6 : TCC Obligation

  % proof status :-)
```

```
  % Search_Pre_TCC7 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC8 : TCC Obligation

  % proof status :-)
  % Search_Pre_TCC9 : TCC Obligation

  Search_Pre : PRED[State] =
    Lambda (s : State) :
      plain_memory?(pm) And
      pm`states(s) And
      % Parameters
        % 1. first and last are allocated in blessed memory
        in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
                                  union(pm`ro_addr, pm`rw_addr)) And
        in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
                                  union(pm`ro_addr, pm`rw_addr)) And
        % 2. the memory locations at first and last store valid pointer values
        valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) And
        valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) And
      % Local Variables
        % 3. current is allocated in blessed writable memory
        in_blessed_memory?(dt_cv_pointer(pointer(uint)),
                                current, pm`rw_addr) And
        % 4. current is disjoint from first
        blocks_disjoint?(first, size_of(pointer(uint)),
                                current, size_of(pointer(uint))) And
        blocks_disjoint?(current, size_of(pointer(uint)),
                                first, size_of(pointer(uint))) And
        % 5. current is disjoint from last
        blocks_disjoint?(last, size_of(pointer(uint)),
                                current, size_of(pointer(uint))) And
        blocks_disjoint?(current, size_of(pointer(uint)),
                                last, size_of(pointer(uint))) And
      % Assertions
      Let first_value =
            data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)),
          last_value =
            data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) In
        % 6. first and last are non null
        not_null?(pointer(uint))(first_value) And
        not_null?(pointer(uint))(last_value) And
      % Array
        % 7. array elements are in blessed memory
        (Forall (i : below[N]) :
           in_blessed_memory?[(range(uint))](dt(uint),
                 down(address_of(pointer(uint))
                           (add(pointer(uint))(first_value, i * size_of(uint)))),
                 union(pm`ro_addr, pm`rw_addr))) And
        % 8. array elements are disjoint from current
        (Forall (i : below[N]) :
```

```
        blocks_disjoint?(
          down(address_of(pointer(uint))
                    (add(pointer(uint))(first_value, i * size_of(uint)))),
          size_of(uint),
          current,
          size_of(pointer(uint)))) And
      (Forall (i : below[N]) :
        blocks_disjoint?(
            current, size_of(pointer(uint)),
            down(address_of(pointer(uint))
                    (add(pointer(uint))(first_value, i * size_of(uint)))),
            size_of(uint))) And
      % 9. array elements contain valid integers
      (Forall (i : below[N]) :
        valid_in_mem(pm, dt(uint))
            (down(address_of(pointer(uint))
              (add(pointer(uint))(first_value, i * size_of(uint))))(s))
      And
      % 10. array bounds
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) =
                  add(pointer(uint))(first_value, N * size_of(uint)) And
      % 11. same array
      same_array(pointer(uint))(first_value, last_value) And
      (Forall (i : below[N]) :
        same_array(pointer(uint))(first_value,
          add(pointer(uint))(first_value, i * size_of(uint))))

% /**
% * search the array element whose value is val
% *
% * @param val the value to search for
% * @param first pointer to the first element of the array (from which
% * to start the search)
% * @param last pointer beyond the last element of the array
% * @return pointer to the first occurrence of val after (including) first
% */
% uint * search(uint val, uint * first, uint * last) {
% assert(first);
% assert(last);

% 1) uint * current = first;

% 2) while(current < last) {
% 3) if (*current == val)
% 4) break;

% 5) ++current;
% }

% proof status :-)
% search_TCC1 : TCC Obligation
```

```
  % proof status :-)
  % search_TCC2 : TCC Obligation

  % proof status :-)
  % search_TCC3 : TCC Obligation

  % proof status :-)
  % search_TCC4 : TCC Obligation

  search : ST[State] =
    % 1)
      e2s[State, Address]
        (assign(pm, dt_cv_pointer(pointer(uint)))
                (id(current),
                 l2r(pm, dt_cv_pointer(pointer(uint)))(id(first)))) ##
    % 2)
      while(N + 1,
            lt_ptr(pointer(uint))
                 (l2r(pm, dt_cv_pointer(pointer(uint)))(id(current)),
                          l2r(pm, dt_cv_pointer(pointer(uint)))(id(last))),
    % 3)
          if_else(
            eq(l2r(pm, dt(uint))(deref(pm, pointer(uint))
                   (l2r(pm, dt_cv_pointer(pointer(uint)))(id(current)))),
              literal(value)),
    % 4)
            break,
            skip) ##
    % 5)
          e2s[State, Address](preinc_ptr(pm, pointer(uint))(id(current)))

  % Postconditions:
  %=================

  % read first after search
  read_first : PRED[State] =
    Lambda (s : State) :
      OK?(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))

  % read current after search
  read_current : PRED[State] =
    Lambda (s : State) :
      OK?(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)) And
      not_null?(pointer(uint))(data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)))

  % proof status :-)
  % deref_current_TCC1 : TCC Obligation

  % proof status :-)
  % deref_current_TCC2 : TCC Obligation

  % read current after search
```

```
deref_current : PRED[State] =
  Lambda (s : State) :
    read_current(s) And
    Let current_value =
          data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)) In
      not_null?(pointer(uint))(current_value) And
      OK?(read_data(pm, dt(uint))
                  (down(address_of(pointer(uint))(current_value)))(s))

% read last after search
read_last : PRED[State] =
  Lambda (s : State) :
    OK?(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s))

% proof status :-)
% read_array_TCC1 : TCC Obligation

% proof status :-)
% read_array_TCC2 : TCC Obligation

% proof status :-)
% read_array_TCC3 : TCC Obligation

% read array after search
read_array : PRED[State] =
  Lambda (s : State) :
    read_first(s) And
    Let first_value =
          data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) In
      not_null?(pointer(uint))(first_value)
    And
      Forall (i : below[N]) :
        OK?(read_data(pm, dt(uint))
              (down(address_of(pointer(uint))
                    (add(pointer(uint))(first_value, i * size_of(uint)))))(s))

% proof status :-)
% not_found_TCC1 : TCC Obligation

% proof status :-)
% not_found_TCC2 : TCC Obligation

% proof status :-)
% not_found_TCC3 : TCC Obligation

% proof status :-)
% not_found_TCC4 : TCC Obligation

% not found:
% the value could not be found in the array; current = last
not_found : PRED[State] =
  Lambda (s : State) :
```

```
      read_first(s) And
      read_current(s) And
      read_last(s) And
      read_array(s) And
      (Let first_value =
            data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) In
        Forall (i : below[N]) :
          (data(read_data(pm, dt(uint))
                 (down(address_of(pointer(uint))(add(pointer(uint))
                       (first_value, i * size_of(uint)))))(s)) /= value))

        Implies
          data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) =
            data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))

% proof status :-)
% found_TCC1 : TCC Obligation

% proof status :-)
% found_TCC2 : TCC Obligation

% proof status :-)
% found_TCC3 : TCC Obligation

found : PRED[State] =
  Lambda (s : State) :
   (read_first(s) And
    read_current(s) And
    read_last(s) And
    read_array(s) And
    deref_current(s) And
    data(read_data(pm, dt_cv_pointer(pointer(uint)))
                   (last)(s)) /=
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)))
  Implies
    data(read_data(pm, dt(uint))
                   (down(address_of(pointer(uint))(
      data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s)))))(s))
        = value

% Lemmas
%========

% proof status :-)
% uchar_range_values_TCC1 : TCC Obligation

% proof status :-)
uchar_range_values : Lemma
  Forall (n : nat) : n <= 255 Implies range(uchar)(n)

% proof status :-)
search_terminates : Lemma
```

```
  Valid(
    Search_Pre,
    search,
    Lambda (s : State) : true)

% proof status :-)
search_read_first : Lemma
  valid(
    Search_Pre,
    search,
    read_first)

% proof status :-)
search_read_current : Lemma
  valid(
    Search_Pre,
    search,
    read_current)

% proof status :-)
search_deref_current : Lemma
  valid(
    Search_Pre,
    search,
    deref_current)

% proof status :-)
search_read_last : Lemma
  valid(
    Search_Pre,
    search,
    read_last)

% proof status :-)
search_read_array : Lemma
  valid(
    Search_Pre,
    search,
    read_array)

% proof status :-)
search_not_found : Lemma
  valid(
    Search_Pre,
    search,
    not_found)

% proof status :-)
search_found : Lemma
  valid(
    Search_Pre,
    search,
```

found)

End Search_Example

B.29 statement-rewrites.pvs

```
% $Id: statement-rewrites.pvs,v 1.10.4.1.4.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Marcus Völp
%
% Rewriting rules for C++ statements. See statements.pvs for the statement
% formalization.
%

% The statement rewriting rules simplify code by symbolic
% simlification of the control flow constructs. For example a break will
% remove all following statements until it reaches a catch break in
% which case break ## catch_break is eleminated from the code.
% Statements are simplified right to left. Here one expression
% transformer respectively one lexpr transformer (StmtResult ->
% ExprResult) is allowed to occur after the simplification point. The
% forward rules simplify these two cases plus those cases with a leading
% statement automatically with the help of the respective rules for the
% standalone case. Because of the requirement on rewriting rules these
% lemmas have to be defined for any form a standalone simplifcation may
% assume. The forms are denoted by the s-c->s, ... sequence which stands
% for statement1 ## complex statement simplifies to statement1.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Single_Statement_Rewrites0[State : Type] : THEORY
BEGIN

  IMPORTING Statements

  stmt : VAR [State -> StmtResult[State]]
  stmt1 : VAR [State -> StmtResult[State]]
  stmt2 : VAR [State -> StmtResult[State]]
  stmt3 : VAR [State -> StmtResult[State]]
  cstmt : VAR [StmtResult[State] -> StmtResult[State]]
  cstmt1 : VAR [StmtResult[State] -> StmtResult[State]]
  fcstmt : VAR [[State -> StmtResult[State]] ->
                [StmtResult[State] -> StmtResult[State]]]

  s : VAR State
  n : VAR int

  % Switch / Default / Continue / Return elemination
  %=================================================

  % Break
```

```
%-------

% proof status :-)
break_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## break ## catch_break)(s) = stmt(s)

% proof status :-)
break_break_stmt : Lemma
  Break?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_catch_continue : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_continue ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_catch_default : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_default ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_lift_stmt : Lemma
  Break?(stmt(s)) Implies
    (stmt ## lift(stmt1) ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_case : Lemma
  Break?(stmt(s)) Implies
    (stmt ## case(n) ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break_default : Lemma
  Break?(stmt(s)) Implies
    (stmt ## default ## catch_break)(s) = (stmt ## catch_break)(s)

% proof status :-)
break_break : Lemma
  Break?(break(s))

% proof status :-)
stmt_break_break : Lemma
  OK?(stmt(s)) Implies
    Break?((stmt ## break)(s))

% proof status :-)
break_stmt_break : Lemma
  Break?(stmt(s)) Implies
    Break?((stmt ## stmt1)(s))

% proof status :-)
```

```
  break_stmt_catch_continue : Lemma
    Break?(stmt(s)) Implies
      Break?((stmt ## catch_continue)(s))

  % proof status :-)
  break_stmt_catch_default : Lemma
    Break?(stmt(s)) Implies
      Break?((stmt ## catch_default)(s))

  % proof status :-)
  break_stmt_lift : Lemma
    Break?(stmt(s)) Implies
      Break?((stmt ## lift(stmt1))(s))

  % proof status :-)
  break_stmt_case : Lemma
    Break?(stmt(s)) Implies
      Break?((stmt ## case(n))(s))

  % proof status :-)
  break_stmt_default : Lemma
    Break?(stmt(s)) Implies
      Break?((stmt ## default)(s))

  % Continue
  %----------
  % proof status :-)
  continue_catch_continue : Lemma
    OK?(stmt(s)) Implies
      (stmt ## continue ## catch_continue)(s) = stmt(s)

  % proof status :-)
  continue_break_stmt : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## stmt1 ## catch_continue)(s) = (stmt ## catch_continue)(s)

  % proof status :-)
  continue_break_catch_continue : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## catch_break ## catch_continue)(s) = (stmt ## catch_continue)(s)

  % proof status :-)
  continue_break_catch_default : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## catch_default ## catch_continue)(s) = (stmt ## catch_continue)(s)

  % proof status :-)
  continue_break_lift_stmt : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## lift(stmt1) ## catch_continue)(s) = (stmt ## catch_continue)(s)

  % proof status :-)
```

```
continue_break_case : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## case(n) ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :-)
continue_break_default : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## default ## catch_continue)(s) = (stmt ## catch_continue)(s)

% proof status :-)
continue_continue : Lemma
  Continue?(continue(s))

% proof status :-)
stmt_continue_continue : Lemma
  OK?(stmt(s)) Implies
    Continue?((stmt ## continue)(s))

% proof status :-)
continue_stmt_continue : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## stmt1)(s))

% proof status :-)
continue_stmt_catch_break : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## catch_break)(s))

% proof status :-)
continue_stmt_catch_default : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## catch_default)(s))

% proof status :-)
continue_stmt_lift : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## lift(stmt1))(s))

% proof status :-)
continue_stmt_case : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## case(n))(s))

% proof status :-)
continue_stmt_default : Lemma
  Continue?(stmt(s)) Implies
    Continue?((stmt ## default)(s))

% switch / default are free-standing ; use fwd rewriting
%=====================================================

% switch
```

```
%---------
% proof status :-)
switch_stmt : Lemma
  switch_result(n) ## stmt = switch_result(n)

% proof status :-)
stmt_switch_stmt : Lemma
  stmt ## switch_result(n) ## stmt1 = stmt ## switch_result(n)

% proof status :-)
switch_case_taken : Lemma
  (switch_result[State](n) ## case(n)) = skip

% proof status :-)
stmt_switch_case_taken : Lemma
  OK?(stmt(s)) Implies
    (stmt ## switch_result(n) ## case(n))(s) = stmt(s)

% proof status :-)
switch_case_not_taken : Lemma
  Forall (n1, n2 : int) : n1 /= n2 Implies
    switch_result[State](n1) ## case(n2) = switch_result(n1)

% proof status :-)
stmt_switch_case_not_taken : Lemma
  Forall (n1, n2 : int) : OK?(stmt(s)) And n1 /= n2 Implies
    (stmt ## switch_result(n1) ## case(n2))(s) = (stmt ## switch_result(n1))(s)

% proof status :-)
switch_default : Lemma
  switch_result[State](n) ## default = switch_result(n)

% proof status :-)
stmt_switch_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## switch_result(n) ## default)(s) = (stmt ## switch_result(n))(s)

% proof status :-)
switch_catch_break : Lemma
  switch_result[State](n) ## catch_break = switch_result(n)

% proof status :-)
stmt_switch_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## switch_result(n) ## catch_break)(s) = (stmt ## switch_result(n))(s)

% proof status :-)
switch_catch_continue : Lemma
  switch_result[State](n) ## catch_continue = switch_result(n)

% proof status :-)
stmt_switch_catch_continue : Lemma
```

```
  OK?(stmt(s)) Implies
    (stmt ## switch_result(n) ## catch_continue)(s) = (stmt ## switch_result(n))(s)

% proof status :-)
switch_catch_default : Lemma
  switch_result[State](n) ## catch_default = switch_result(n)

% proof status :-)
stmt_switch_catch_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## switch_result(n) ## catch_default)(s) = (stmt ## switch_result(n))(s)

% default
%----------
% proof status :-)
default_stmt : Lemma
  default_result ## stmt = default_result

% proof status :-)
stmt_default_stmt : Lemma
  stmt ## default_result ## stmt1 = stmt ## default_result

% proof status :-)
default_case : Lemma
  default_result[State] ## case(n) = default_result

% proof status :-)
stmt_default_case : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## case(n))(s) = (stmt ## default_result)(s)

% proof status :-)
default_default : Lemma
  default_result[State] ## default = skip

% proof status :-)
stmt_default_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## default)(s) = stmt(s)

% proof status :-)
default_catch_break : Lemma
  default_result[State] ## catch_break = default_result

% proof status :-)
stmt_default_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## catch_break)(s) = (stmt ## default_result)(s)

% proof status :-)
default_catch_continue : Lemma
  default_result[State] ## catch_continue = default_result
```

```
% proof status :-)
stmt_default_catch_continue : Lemma
  OK?(stmt(s)) Implies
  (stmt ## default_result ## catch_continue)(s) = (stmt ## default_result)(s)

% proof status :-)
default_catch_default : Lemma
  default_result[State] ## catch_default = skip

% proof status :-)
stmt_default_catch_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## default_result ## catch_default)(s) = stmt(s)

% Return rewrites
%==================

% proof status :-)
return_void_result : Lemma
  return_void[State] = return_result


% Skip Elemination
%==================

% proof status :-)
skip_ok : Lemma
  OK?(skip(s))

% proof status :-)
% skip_state_TCC1 : TCC Obligation

% proof status :-)
skip_state : Lemma
  state(skip(s)) = s

% proof status :-)
skip_elemination : Lemma
  (stmt ## skip) = stmt

% proof status :-)
stmt_skip_elemination : Lemma
  (skip ## stmt) = stmt

% Stmt Ok Elemination
%====================

% proof status :-)
stmt_remove_catch_default : Lemma
  OK?(stmt(s)) Implies
    (stmt ## catch_default)(s) = stmt(s)
```

```
% proof status :-)
stmt_remove_case : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## case(n))(s) = stmt1(s)

% proof status :-)
stmt_remove_default : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## default)(s) = stmt1(s)

% proof status :-)
stmt_remove_catch_continue : Lemma
  OK?(stmt(s)) Implies
    (stmt ## catch_continue)(s) = stmt(s)

% proof status :-)
stmt_remove_catch_break : Lemma
  OK?(stmt(s)) Implies
    (stmt ## catch_break)(s) = stmt(s)

% proof status :-)
stmt_ok_lift : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## lift(stmt2))(s) = (stmt1 ## stmt2)(s)

% While
%=======

b_ex : VAR [State -> ExprResult[State, bool]]
max : Var nat

% proof status :-)
% while_unroll_TCC1 : TCC Obligation

% proof status :-)
while_unroll : Lemma
  max > 0 Implies
    while(max, b_ex, stmt)(s) =
    if_else(b_ex, stmt ## catch_continue ##
      while(max - 1, b_ex, stmt) ## catch_break, skip)(s)

% proof status :-)
stmt_while_unroll : Lemma
  max > 0 Implies
    (stmt1 ## while(max, b_ex, stmt2))(s) =
    (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ##
      while(max - 1, b_ex, stmt2) ## catch_break, skip))(s)

% Condition universally true in all states implies hang of while
% proof status :-)
iterate_while_ok : Lemma
```

```
    Forall (n : nat) :
      (Forall (s : State) : OK?((e2s(b_ex) ## stmt)(s)))
    Implies
      OK?(iterate_while(n, b_ex, stmt)(s))

  % proof status :-)
  while_hang : Lemma
    (Forall (s : State) :
      OK?(b_ex(s)) And data(b_ex(s)) And
      OK?((e2s(b_ex) ## stmt)(s)))
  Implies
    while(b_ex, stmt) = hang_result

  % Composition
  %=============

  % proof status :-)
  composition_assoc_rewrite_stmt_ok : Lemma
    OK?(stmt(s)) Implies
    (stmt ## (stmt1 ## cstmt))(s) = ((stmt ## stmt1) ## cstmt)(s)

  % proof status :-)
  composition_assoc_rewrite_stmt_ok_cstmt : Lemma
    OK?(stmt(s)) Implies
    ((stmt ## (stmt1 ## cstmt)) ## cstmt1)(s) = (((stmt ## stmt1) ## cstmt) ## cstmt1)(s)


END Single_Statement_Rewrites0

Single_Statement_Rewrites1[State, Data : Type] : THEORY
BEGIN

  Importing Single_Statement_Rewrites0, Plain_Mem_Rewrites

  stmt : VAR [State -> StmtResult[State]]
  stmt1 : VAR [State -> StmtResult[State]]
  stmt2 : VAR [State -> StmtResult[State]]
  stmt3 : VAR [State -> StmtResult[State]]
  cstmt : VAR [StmtResult[State] -> StmtResult[State]]
  cstmt1 : VAR [StmtResult[State] -> StmtResult[State]]
  fcstmt : VAR [[State -> StmtResult[State]] ->
                [StmtResult[State] -> StmtResult[State]]]
  fstmt : VAR [Data -> [State -> StmtResult[State]]]
  expr : VAR [State -> ExprResult[State, Data]]
  lexpr : VAR [StmtResult[State] -> ExprResult[State, Data]]

  data : VAR Data

  s : VAR State
  n : VAR int

  % Stmt Ok Elemination Expr Forms
```

```
%|=============================|
% proof status :-)
forward_s_c_s_of_s_expr : Lemma
  (stmt ## cstmt)(s) = stmt(s) Implies
    (stmt ## cstmt ## expr)(s) = (stmt ## expr)(s)

% proof status :-)
stmt_ok_lift_expr : Lemma
  OK?(stmt1(s)) Implies
    (stmt1 ## lift(stmt2) ## expr)(s) = (stmt1 ## stmt2 ## expr)(s)

% proof status :-)
break_break_stmt_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_catch_continue_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_continue ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_catch_default_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## catch_default ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_lift_stmt_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## lift(stmt1) ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_case_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## case(n) ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
break_break_default_expr : Lemma
  Break?(stmt(s)) Implies
    (stmt ## default ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
continue_break_stmt_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## stmt1 ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

% proof status :-)
continue_break_catch_continue_expr : Lemma
  Continue?(stmt(s)) Implies
    (stmt ## catch_break ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

% proof status :-)
```

```
  continue_break_catch_default_expr : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## catch_default ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

  % proof status :-)
  continue_break_lift_stmt_expr : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## lift(stmt1) ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

  % proof status :-)
  continue_break_case_expr : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## case(n) ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

  % proof status :-)
  continue_break_default_expr : Lemma
    Continue?(stmt(s)) Implies
      (stmt ## default ## catch_continue ## expr)(s) = (stmt ## catch_continue ## expr)(s)

  % While
  %=======

  b_ex : VAR [State -> ExprResult[State, bool]]
  max : Var nat

  % proof status :-)
  % while_unroll_expr_TCC1 : TCC Obligation

  % proof status :-)
  while_unroll_expr : Lemma
    max > 0 Implies
      (while(max, b_ex, stmt) ## expr)(s) =
      (if_else(b_ex, stmt ## catch_continue ##
        while(max - 1, b_ex, stmt) ## catch_break, skip) ## expr)(s)

  % proof status :-)
  while_unroll_lexpr : Lemma
    max > 0 Implies
      (while(max, b_ex, stmt) ## lexpr)(s) =
      (if_else(b_ex, stmt ## catch_continue ##
        while(max - 1, b_ex, stmt) ## catch_break, skip) ## lexpr)(s)

  % proof status :-)
  stmt_while_unroll_expr : Lemma
    max > 0 Implies
      (stmt1 ## while(max, b_ex, stmt2) ## expr)(s) =
      (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ##
        while(max - 1, b_ex, stmt2) ## catch_break, skip) ## expr)(s)

  % proof status :-)
  stmt_while_unroll_lexpr : Lemma
    max > 0 Implies
```

```
    (stmt1 ## while(max, b_ex, stmt2) ## lexpr)(s) =
    (stmt1 ## if_else(b_ex, stmt2 ## catch_continue ##
      while(max - 1, b_ex, stmt2) ## catch_break, skip) ## lexpr)(s)

% E2s-Final Rewrite
%====================
% proof status :-)
e2s_ok : Lemma
  OK?(expr(s)) Implies
    OK?(e2s(expr)(s))

% proof status :-)
stmt_e2s_ok : Lemma
  OK?((stmt ## expr)(s)) Implies
    OK?((stmt ## e2s(expr))(s))

% proof status :-)
% e2s_state_TCC1 : TCC Obligation

% proof status :-)
% e2s_state_TCC2 : TCC Obligation

% proof status :-)
e2s_state : Lemma
  OK?(expr(s)) Implies
    state(e2s(expr)(s)) = state(expr(s))

% proof status :-)
% stmt_e2s_state_TCC1 : TCC Obligation

% proof status :-)
% stmt_e2s_state_TCC2 : TCC Obligation

% proof status :-)
stmt_e2s_state : Lemma
  OK?((stmt ## expr)(s)) Implies
    state((stmt ## e2s(expr))(s)) = state((stmt ## expr)(s))

% Fexpr / Fstmt Rewrite
%======================

% proof status :-)
comp_eval_if_ok_fstmt : Lemma
  OK?(expr(s)) Implies
    (expr ## fstmt)(s) = (e2s(expr) ## fstmt(data(expr(s))))(s)

% proof status :-)
stmt_eval_if_ok_fstmt : Lemma
  OK?((stmt ## expr)(s))
Implies
  (stmt ## (expr ## fstmt))(s) = (stmt ## (e2s(expr) ## fstmt(data((stmt ## expr)(s)))))(s)
```

```
% proof status :-)
comp_eval_if_ok_fstmt_cstmt : Lemma
  OK?(expr(s)) Implies
    ((expr ## fstmt) ## cstmt)(s) = (e2s(expr) ## fstmt(data(expr(s))) ## cstmt)(s)

% proof status :-)
stmt_eval_if_ok_fstmt_cstmt : Lemma
  OK?((stmt ## expr)(s))
Implies
  (stmt ## (expr ## fstmt) ## cstmt)(s) = (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## cstm

% Composition
%=============

% proof status :-)
composition_assoc_rewrite_expr_1 : Lemma
  (stmt ## (stmt1 ## expr)) = ((stmt ## stmt1) ## expr)

% proof status :-)
composition_assoc_rewrite_expr_2 : Lemma
  (stmt ## (cstmt ## expr)) = ((stmt ## cstmt) ## expr)

% proof status :-)
composition_assoc_rewrite_expr_3 : Lemma
  (cstmt ## (cstmt1 ## expr)) = ((cstmt ## cstmt1) ## expr)

% proof status :-)
composition_assoc_rewrite_expr_4 : Lemma
  (stmt ## (lift(stmt1) ## lexpr)) = ((stmt ## stmt1) ## lexpr)

% proof status :-)
composition_assoc_rewrite_expr_5 : Lemma
  (stmt ## (cstmt ## lexpr)) = ((stmt ## cstmt) ## lexpr)

% proof status :-)
composition_assoc_rewrite_expr_6 : Lemma
  (cstmt ## (cstmt1 ## lexpr)) = ((cstmt ## cstmt1) ## lexpr)

% proof status :-)
composition_assoc_rewrite_stmt_ok : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt))(s) = ((stmt ## stmt1) ## cstmt)(s)

% proof status :-)
composition_assoc_rewrite_stmt_ok_expr : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt) ## expr)(s) = ((stmt ## stmt1) ## cstmt ## expr)(s)

% proof status :-)
composition_assoc_rewrite_stmt_ok_expr_lexpr : Lemma
  OK?(stmt(s)) Implies
  (stmt ## (stmt1 ## cstmt) ## lexpr)(s) = ((stmt ## stmt1) ## cstmt ## lexpr)(s)
```

```
  % proof status :-)
  ok_result_fstmt : Lemma
    Forall (data : Data) :
      ok_result(data) ## fstmt = fstmt(data)

  pm : Var Plain_Memory[State]
  dt : Var (interpreted_data_type?[Data])
  addr : Var Address

  % proof status :-)
  pm_q_prop_read_ok_s_read_ok : Lemma
    pm_q_prop_read(pm, dt, addr)(skip(s)) Implies
      OK?(read_data(pm, dt)(addr)(s))


END Single_Statement_Rewrites1

Single_Statement_Rewrites2 [State, Data1, Data2 : Type] : THEORY
BEGIN

  Importing Single_Statement_Rewrites1

  s : VAR State

  stmt : VAR [State -> StmtResult[State]]
  cstmt : VAR [StmtResult[State] -> StmtResult[State]]
  expr : VAR [State -> ExprResult[State, Data1]]
  expr1 : VAR [State -> ExprResult[State, Data2]]
  fexpr : VAR [Data2 -> [State -> ExprResult[State, Data1]]]
  fstmt : VAR [Data1 -> [State -> StmtResult[State]]]

  % Lemmas to speed up rewriting
  %=============================
  % proof status :-)
  ok_result_fexpr : Lemma
    Forall (data : Data2) :
      ok_result(data) ## fexpr = fexpr(data)

  % E2s Expr
  %==========

  % proof status :-)
  e2s_merge : Lemma
    (e2s(expr) ## e2s(expr1)) = e2s(expr ## expr1)

  % proof status :-)
  stmt_e2s_merge : Lemma
    (stmt ## e2s(expr) ## e2s(expr1)) =
      (stmt ## e2s(expr ## expr1))

  % proof status :-)
```

```
  e2s_expr : Lemma
    (e2s(expr1) ## expr)(s) = (expr1 ## expr)(s)

  % proof status :-)
  stmt_e2s_expr : Lemma
    (stmt ## e2s(expr1) ## expr)(s) = (stmt ## expr1 ## expr)(s)

  % proof status :-)
  composition_assoc_expression_rewrite_stmt : Lemma
    (stmt ## (expr ## expr1)) = ((stmt ## expr) ## expr1)

  % proof status :-)
  comp_eval_if_ok_fstmt_expr : Lemma
    OK?(expr(s)) Implies
      ((expr ## fstmt) ## expr1)(s) = (e2s(expr) ## fstmt(data(expr(s))) ## expr1)(s)

  % proof status :-)
  stmt_eval_if_ok_fstmt_expr : Lemma
    OK?((stmt ## expr)(s)) Implies
      (stmt ## (expr ## fstmt) ## expr1)(s) =
      (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## expr1)(s)

  % proof status :-)
  stmt_eval_if_ok_fexpr : Lemma
    OK?((stmt ## expr1)(s)) Implies
      (stmt ## (expr1 ## fexpr))(s) = (stmt ## expr1 ## fexpr(data((stmt ## expr1)(s))))(s)

  % proof status :-)
  comp_eval_if_ok_fstmt_cstmt_expr : Lemma
    OK?(expr(s)) Implies
      (((expr ## fstmt) ## cstmt) ## expr1)(s) =
      (((e2s(expr) ## fstmt(data(expr(s)))) ## cstmt) ## expr1)(s)

  % proof status :-)
  stmt_eval_if_ok_fstmt_cstmt_expr : Lemma
    OK?((stmt ## expr)(s)) Implies
      (stmt ## (expr ## fstmt) ## cstmt ## expr1)(s) =
      (stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## cstmt ## expr1)(s)

End Single_Statement_Rewrites2


Single_Statement_Rewrites3 [State, Data1, Data2, Data3 : Type] : THEORY
BEGIN

  Importing Single_Statement_Rewrites2

  % Assumption: there is not always an expression at the end !!!

  expr : VAR [State -> ExprResult[State, Data1]]
  expr1 : VAR [State -> ExprResult[State, Data2]]
  expr2 : VAR [State -> ExprResult[State, Data3]]
```

```
  fexpr : VAR [Data1 -> [State -> ExprResult[State, Data3]]]

  stmt : VAR [State -> StmtResult[State]]

  s : VAR State

  % E2s Expr
  %==========

  % proof status :-)
  ok_result_elemination_2 : Lemma
    Forall (data : Data3) :
      ((expr ## ok_result[State, Data3](data)) ## expr1) = (expr ## expr1)

  % Composition
  %============

  % proof status :-)
  comp_eval_if_ok_fexpr_expr : Lemma
    OK?(expr(s)) Implies
      ((expr ## fexpr) ## expr1)(s) = (expr ## fexpr(data(expr(s))) ## expr1)(s)

  % proof status :-)
  expr_eval_if_ok_fexpr : Lemma
    OK?((expr1 ## expr)(s)) Implies
      (expr1 ## (expr ## fexpr))(s) = (expr1 ## expr ## fexpr(data((expr1 ## expr)(s))))(s)

  % proof status :-)
  composition_assoc_expression_rewrite : Lemma
    (expr ## (expr1 ## expr2)) = ((expr ## expr1) ## expr2)

  % proof status :-)
  stmt_eval_if_ok_fexpr_expr : Lemma
    OK?((stmt ## expr)(s)) Implies
      (stmt ## (expr ## fexpr) ## expr1)(s) =
        (stmt ## expr ## fexpr(data((stmt ## expr)(s))) ## expr1)(s)

END Single_Statement_Rewrites3


Single_Statement_Rewrites4 [State, Data1, Data2, Data3, Data4 : Type] : THEORY
BEGIN

  Importing Single_Statement_Rewrites3

  expr : VAR [State -> ExprResult[State, Data1]]
  expr1 : VAR [State -> ExprResult[State, Data2]]
  expr2 : VAR [State -> ExprResult[State, Data4]]
  fexpr : VAR [Data2 -> [State -> ExprResult[State, Data3]]]

  s : VAR State
```

```
  % Composition
  %=============

  % proof status :-)
  expr_eval_if_ok_fexpr_expr : Lemma
    OK?((expr ## expr1)(s)) Implies
      (expr ## (expr1 ## fexpr) ## expr2)(s) =
        (expr ## expr1 ## fexpr(data((expr ## expr1)(s))) ## expr2)(s)

END Single_Statement_Rewrites4

While_Hoare[State, Ord : Type] : Theory
Begin

  Importing Single_Statement_Rewrites4, More_Relations

  s0 : Var State
  < : Var PRED[[Ord, Ord]]

  b_ex : Var [State -> ExprResult[State, bool]]
  body : Var [State -> StmtResult[State]]

  n : Var nat

  % proof status :-)
  iterate_while_right : Lemma
    Forall (n : nat) :
      iterate_while(1 + n, b_ex, body) =
        iterate_while(n, b_ex, body) ## lift(e2s(b_ex) ## body ## catch_continue)

  % While Invariant
  %-----------------

  % proof status :-)
  % while_variant?_TCC1 : TCC Obligation

  while_variant?(b_ex, body, <)(variant : [State -> Ord],
                    Q : PRED[StmtResult[State]]) : bool =
      well_founded?(<)
    And
      while_invariant?(b_ex, body)(Q, Q)
    And
      Forall(s : State) :
        Q(OK(s)) And OK?(b_ex(s)) And data(b_ex(s)) And
        OK?((e2s(b_ex) ## body ## catch_continue)(s))
      Implies
        variant(state((e2s(b_ex) ## body ## catch_continue)(s))) < variant(s)


  % proof status :-)
  % while_no_cb_TCC1 : TCC Obligation
```

```
while_no_cb(b_ex, body) : [State -> StmtResult[State]] =
  Lambda (s : State) :
    If Exists (n : nat): while_termination_point?(b_ex, body, s)(n) Then
      Let
        i = min[nat](while_termination_point?(b_ex, body, s))
      In
        (iterate_while(i, b_ex, body) ## e2s(b_ex))(s)
    Else
      Hang
    Endif


% proof status :-)
while_no_cb_cb_while : Lemma
  while(b_ex, body) = (while_no_cb(b_ex, body) ## catch_break)


% proof status :-)
iterate_while_invariant : Lemma
  Forall (inv_abnorm?, inv_norm? : PRED[StmtResult[State]], n : nat) :
    while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?)
  And
    inv_abnorm?(OK(s0))
  And
    (Forall (m : nat) : m < n Implies Not while_termination_point?(b_ex, body, s0)(m))
  Implies
    inv_abnorm?(iterate_while(n, b_ex, body)(s0))


% proof status :-)
min_termination_point_no_less : Lemma
  Forall (m : nat) :
    (Exists (n : nat) : while_termination_point?(b_ex, body, s0)(n))
  And
    m < min[nat](while_termination_point?(b_ex, body, s0))
  Implies
    Not while_termination_point?(b_ex, body, s0)(m)


% proof status :-)
while_terminates : Lemma
  Forall (term_inv? : PRED[StmtResult[State]], variant : [State -> Ord]) :
    while_variant?(b_ex, body, <)(variant, term_inv?) And %term_inv?(s0)
    term_inv?(OK(s0))
  Implies
    Exists (n : nat) : while_termination_point?(b_ex, body, s0)(n)


% proof status :-)
hoare_while : Lemma
  Forall (term_inv? : PRED[StmtResult[State]],
```

```
            data_inv_abnorm?, data_inv_norm? : PRED[StmtResult[State]],
            variant : [State -> Ord]) :
      while_variant?(b_ex, body, <)(variant, term_inv?)
    And
      while_invariant?(b_ex, body)(data_inv_abnorm?, data_inv_norm?)
    And
      data_inv_abnorm?(OK(s0)) And %term_inv?(s0)
      term_inv?(OK(s0))
    Implies
      Let res = while_no_cb(b_ex, body)(s0) In
        term_inv?(res)
      And
        ((data_inv_abnorm?(res) And Not OK?(res)) OR
         data_inv_norm?(res))

End While_Hoare

While_Rewrites[State, Ord : Type] : Theory
Begin

  Importing While_Hoare

  s : VAR State
  pm : Var Plain_Memory[State]

  b_ex : VAR [State -> ExprResult[State, bool]]
  body : VAR [State -> StmtResult[State]]

  stmt : VAR [State -> StmtResult[State]]

  % data invariant: P = abnormal termination, Q = normal termination
  % termination invariant : Q
  P, Q, R : VAR [State -> PRED[StmtResult[State]]]

  % property
  S : VAR PRED[StmtResult[State]]

  % variant
  variant : VAR [State -> [State -> Ord]]
  < : VAR PRED[[Ord, Ord]]


  while_no_cb(b_ex, body)(P, Q, R, variant, <) : [State -> StmtResult[State]] =
    while_no_cb[State, Ord](b_ex, body)


  % proof status :-)
  while_to_while_no_cb : Lemma
    while[State, Ord](b_ex, body)(P, Q, R, variant, <) =
      (while_no_cb(b_ex, body)(P, Q, R, variant, <) ## catch_break)
```

```
% proof status :-)
while_inv_rewrite_termination_result : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  (Forall (res : StmtResult[State]) : R(s)(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))


% proof status :-)
while_inv_rewrite_data_ok : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  OK?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
And
  (Forall (res : StmtResult[State]) : Q(s)(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))


% proof status :-)
while_inv_rewrite_data_break : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  Break?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
And
  (Forall (res : StmtResult[State]) : (P(s)(res) Or Q(s)(res)) And Break?(res) Implies S(res))
Implies
  S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))


% proof status :-)
while_inv_rewrite_data_return : Lemma
  P(s)(skip(s)) And R(s)(skip(s))
And
  while_variant?(b_ex, body, <)(variant(s), R(s))
And
  while_invariant?(b_ex, body)(P(s), Q(s))
And
  Return?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
```

```
  And
    (Forall (res : StmtResult[State]) : (P(s)(res) Or Q(s)(res)) And Return?(res) Implies S(res))
  Implies
    S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))


  % proof status :-)
  % stmt_while_inv_rewrite_termination_result_TCC1 : TCC Obligation

  % proof status :-)
  stmt_while_inv_rewrite_termination_result : Lemma
    OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
  And
    while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
  And
    while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
  And
    (Forall (res : StmtResult[State]) : R(state(stmt(s)))(res) Implies S(res))
  Implies
    S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))

% (auto-rewrite "stmt_while_inv_rewrite_data_ok!")
% Can't rewrite using stmt_while_inv_rewrite_data_ok: LHS key S is bad.

  % proof status :-)
  stmt_while_inv_rewrite_data_ok : Lemma
    OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
  And
    while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
  And
    while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
  And
    OK?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))
  And
    (Forall (res : StmtResult[State]) : Q(state(stmt(s)))(res) Implies S(res))
  Implies
    S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))

  % proof status :-)
  stmt_while_inv_rewrite_data_break : Lemma
    OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
  And
    while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
  And
    while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
  And
    Break?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))
  And
    (Forall (res : StmtResult[State]) :
       (P(state(stmt(s)))(res) Or Q(state(stmt(s)))(res)) And Break?(res) Implies S(res))
  Implies
    S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))
```

```
  % proof status :-)
  stmt_while_inv_rewrite_data_return : Lemma
    OK?(stmt(s)) And P(state(stmt(s)))(stmt(s)) And R(state(stmt(s)))(stmt(s))
  And
    while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
  And
    while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s))))
  And
    Return?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))
  And
    (Forall (res : StmtResult[State]) :
       (P(state(stmt(s)))(res) Or Q(state(stmt(s)))(res)) And Return?(res) Implies S(res))
  Implies
    S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))


  % proof status :-)
  pm_q_prop_while : Lemma
    Let pm_q_prop_inv = Lambda (s : State) : Lambda (res : StmtResult[State]) :
                          pm_q_prop(pm)(catch_break(res))
    In
      pm`states(s) And R(s)(skip(s))
    And
      while_variant?(b_ex, body, <)(variant(s), R(s))
    And
      while_invariant?(b_ex, body)(pm_q_prop_inv(s), pm_q_prop_inv(s))
    Implies
      pm_q_prop(pm)(while(b_ex, body)(P, Q, R, variant, <)(s))


  % proof status :-)
  % pm_q_prop_stmt_while_TCC1 : TCC Obligation

  % proof status :-)
  pm_q_prop_stmt_while : Lemma
    Let pm_q_prop_inv = Lambda (s : State) : Lambda (res : StmtResult[State]) :
                          pm_q_prop(pm)(catch_break(res))
    In
      pm_q_prop(pm)(stmt(s)) And R(state(stmt(s)))(stmt(s))
    And
      while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s))))
    And
      while_invariant?(b_ex, body)(pm_q_prop_inv(state(stmt(s))), pm_q_prop_inv(state(stmt(s))))
    Implies
      pm_q_prop(pm)((stmt ## while(b_ex, body)(P, Q, R, variant, <))(s))

End While_Rewrites

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
```

```
% Statement_Rewrites: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Statement_Rewrites[State : Type] : Theory
Begin

  Importing While_Rewrites

End Statement_Rewrites
```

B.30 statements.pvs

```
% $Id: statements.pvs,v 1.37.2.1.2.1.2.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Tjark Weber (based on lecture material by Hendrik Tews)
% Marcus Völp (rewriting rules)
% (c) 2007 Radboud University
%
% A shallow embedding of (some) C++ statements as state transformers.

% Section numbers in comments throughout this file refer to
% "Programming languages − C++", ISO/IEC 14882:1998(E)

% Most statements have type
% State −> StmtResult[State]
%
% "case" and "default" label statements however have type
% StmtResult[State] −> StmtResult[State]
% so that they can catch a "Switch" or "Default" result.
%
% Functions of the former type are lifted to the latter type by modifying
% results of the form
% OK(state)
% only, and passing the result through unchanged otherwise (see the definitions
% of ## in state−transformer.pvs).

% Deviations from the C++ standard:
%
% "goto" (6.6.4) and goto labels (6.1) have not been modeled. This would not
% be difficult in principle, but the possibility of jumps into substatements
% (e.g. into an "if" or "while") would make the formalization of these
% statements more complicated. Also note that (backward) gotos may cause
% nontermination and would need to be modeled somewhat similar to loops.
%
% For the same reason, "case" and "default" labels are required to appear on
% top−level statements contained within the (compound) substatement of a
% "switch" statement only, contrary to what 6.4.2.6 specifies. (Note that this
% rules out code like Duff's Device.)
%
% The condition of a "switch" statement must evaluate to a value of type int
% (cf. 6.4.2.2).
```

```
% >> This is according to the standard: the condition must be of integral type,
% enum type (which has an integral base type) or of a class type that can
% be converted to an integral or enum type.
%
% Functions must always use a "return" statement to return; flowing off the
% end of a function is not allowed (cf. 6.6.3.2). This can be achieved through
% a trivial source code transformation, where a "return_void" statement is
% appended (as the body's last statement) to each function with return type
% void. Alternatively, "catch_return" (see expressions.pvs) could be
% overloaded to convert OK(s) into OK(s, unit), but that would probably cause
% ambiguities in type checking.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.1: Labeled statement
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

LabeledStatements[State : Type] : THEORY
BEGIN

  IMPORTING StmtResult

  const : VAR int
  stmt : VAR [State -> StmtResult[State]]

% res : VAR StmtResult[State]
% ident : VAR string

  % goto target: not modeled
% label(ident)(res) : StmtResult[State] =
% CASES res OF
% Goto(state, label):
% if label = ident then OK(state) else res endif
% ELSE res
% ENDCASES

  % switch: case target
  case(const) : [StmtResult[State] -> StmtResult[State]] =
    Lambda (res : StmtResult[State]) :
      CASES res OF
        Switch(state, case):
          if case = const then OK(state) else res endif
        ELSE res
      ENDCASES

  % switch: default target
  default : [StmtResult[State] -> StmtResult[State]] =
    Lambda (res : StmtResult[State]) :
      CASES res OF
        Default(state): OK(state)
        ELSE res
```

```
    ENDCASES

END LabeledStatements


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.2: Expression statement
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SkipStatements[State : Type] : THEORY
BEGIN

  IMPORTING StmtResult

  s : VAR State

  skip : [State -> StmtResult[State]] =
    Lambda (s : State) : OK(s)

END SkipStatements


ExpressionStatements[State, ExprData : Type] : THEORY
BEGIN

  IMPORTING ExprResult, StmtResult

  s : VAR State
  ex : VAR [State -> ExprResult[State,ExprData]]

  % expression-to-statement
  e2s(ex) : [State -> StmtResult[State]] =
    Lambda (s : State) :
      CASES ex(s) OF
        OK(state, data): OK(state), % forget the expression data
        Fatal: Fatal,
        Hang: Hang,
        % TODO: other possible expression results?
        Exception(ex, state) : Exception(ex, state)
      ENDCASES

END ExpressionStatements

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.3: Compound statement or block
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% There is no need to model blocks here (or so I hope - what about the scope of
```

```
% declarations though?). Composition of statements is defined as
% ##
% in state−transformer.pvs.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Auxiliary catch_XXX functions; used to model jumps (cf. 6.6)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CatchAux[State : Type] : THEORY
BEGIN

  IMPORTING ExprResult, StmtResult

  res : VAR StmtResult[State]

   catch_break(res) : StmtResult[State] =
      CASES res OF
        Break(state): OK(state)
        ELSE res
      ENDCASES

   catch_continue(res) : StmtResult[State] =
      CASES res OF
        Continue(state): OK(state)
        ELSE res
      ENDCASES

  % catch_switch: not needed since a switch statement will be entered with a
  % "Switch" result only if there is a matching case target

  catch_default(res) : StmtResult[State] =
      CASES res OF
        Default(state): OK(state)
        ELSE res
      ENDCASES

  catch_return(res) : StmtResult[State] =
      CASES res OF
        Return(state): OK(state)
        ELSE res
      ENDCASES


END CatchAux


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.4: Selection statements (if, switch)
```

```
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

SelectionStatements[State : Type] : THEORY
BEGIN

  IMPORTING State_Transformer, CatchAux

  s : VAR State

  b_ex : VAR [State -> ExprResult[State,bool]]
  t1, t2: VAR [State -> StmtResult[State]]

  n_ex : VAR [State -> ExprResult[State,int]]
  case_consts : VAR list[int]
  case_stm : VAR [StmtResult[State] -> StmtResult[State]]

  % 6.4.1
  % if ... else ...
  if_else(b_ex, t1, t2) : [State -> StmtResult[State]] =
    (b_ex ## lambda (b : bool): if b then t1 else t2 endif)

  % if without else (6.4.1) is equivalent to if_else(b_ex, t1, skip)

  % 6.4.2
  % switch

  % case_consts is a list of all constants that occur as case labels within the
  % switch statement. We need it to determine beforehand if there is one such
  % label that matches; otherwise the default label (if there is one) must be
  % taken (cf. 6.4.2.5).

  switch(n_ex, case_consts, case_stm) : [State -> StmtResult[State]] =
  (
    n_ex ## (lambda (n : int) :
    if member(n, case_consts) then
      switch_result(n) ## case_stm
    else
      default_result ## case_stm ## catch_default
    endif) ## catch_break
  )

END SelectionStatements


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.5: Iteration statements (while, do while, for)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

IterationStatements1[State : Type] : THEORY
```

```
BEGIN

  IMPORTING SkipStatements, ExpressionStatements,
            SelectionStatements, Number_Props

  s : VAR State

  b_ex: VAR [State -> ExprResult[State,bool]]
  body: VAR [State -> StmtResult[State]]

  % 6.5.1
  % while

  % proof status :-)
  % iterate_while_TCC1 : TCC Obligation

  % proof status :-)
  % iterate_while_TCC2 : TCC Obligation

  % unroll the loop n times (must return the state immediately before
  % evaluation of b_ex, so that we can define while_termination_point?)
  iterate_while(n : nat, b_ex, body)(s) : RECURSIVE
      StmtResult[State] =
    (if n = 0 then
      skip
    else
      e2s(b_ex) ##
      body ##
      catch_continue ##
      iterate_while(n-1, b_ex, body)
    endif)(s)
  MEASURE n

  % proof status :-)
  % while_termination_point?_TCC1 : TCC Obligation

  while_termination_point?(b_ex, body, s)(n : nat) : bool =
    let res = iterate_while(n, b_ex, body)(s) in
      not OK?(res) or
      not OK?(b_ex(state(res))) or
      not data(b_ex(state(res)))

  % proof status :-)
  % while_TCC1 : TCC Obligation

  while(b_ex, body) : [State -> StmtResult[State]] =
    Lambda (s : State) :
      if exists (n : nat): while_termination_point?(b_ex, body, s)(n) then
        let
          i = min[nat](while_termination_point?(b_ex, body, s))
        in (
          iterate_while(i, b_ex, body) ##
```

```
        e2s(b_ex) ## % evaluate b_ex one more time
        catch_break
      )(s)
    else
      Hang
    endif

% proof status :-)
nonzero_min_while_termination_point : Lemma
  (Exists (n : nat) : while_termination_point?(b_ex, body, s)(n)) And
   Not min[nat](while_termination_point?(b_ex, body, s)) = 0
    Implies
  OK?(b_ex(s)) And data(b_ex(s))

% Warning do not auto-rewrite with while_as_if_while.
% Use while(max, ...) and while_unroll instead
% proof status :-)
while_as_if_while : Lemma
  while(b_ex, body) = if_else(b_ex, body ## catch_continue ##
                              while(b_ex, body) ## catch_break, skip)

while(max : nat, b_ex, body) : [State -> StmtResult[State]] =
  while(b_ex, body)

% Warning do not auto-rewrite with this lemma --- takes too long;
% use while_unroll_rewrite* instead
% while_unroll : Lemma
% Forall (max : nat) : max > 0
% Implies
% while(max, b_ex, body) =
% if_else(b_ex, body ## catch_continue ## while(max - 1, b_ex, body) ##
% catch_break, skip)

% while_unroll_0 : Lemma
% while(0, b_ex, body) = while(b_ex, body)


while_invariant?(b_ex, body)(P, Q : PRED[StmtResult[State]]) : bool =
  Forall (s : State) :
    P(skip(s)) Implies
      If OK?(b_ex(s)) And data(b_ex(s)) Then
        P((e2s(b_ex) ## body ## catch_continue[State])(s))
      Else
        Q(e2s(b_ex)(s))
      Endif


% 6.5.2
% do

% unroll the loop n times (must return the state immediately before
% evaluation of b_ex, so that we can define do_termination_point?)
```

```
iterate_do(n : nat, body, b_ex)(s) : RECURSIVE
    StmtResult[State] =
  (if n = 0 then
    body ##
    catch_continue
  else
    body ##
    catch_continue ##
    e2s(b_ex) ##
    iterate_do(n−1, body, b_ex)
  endif)(s)
MEASURE n

% proof status :−)
iterate_do_as_while : Lemma
  Forall (n : nat) :
    iterate_do(n, body, b_ex) = (body ## catch_continue ##
                                   iterate_while(n, b_ex, body))

% proof status :−)
% do_termination_point?_TCC1 : TCC Obligation

do_termination_point?(body, b_ex, s)(n : nat) : bool =
  let res = iterate_do(n, body, b_ex)(s) in
    not OK?(res) or
    not OK?(b_ex(state(res))) or
    not data(b_ex(state(res)))

% proof status :−)
% termination_point_do_as_while_TCC1 : TCC Obligation

% proof status :−)
termination_point_do_as_while : Lemma
  OK?((body ## catch_continue)(s)) Implies
    do_termination_point?(body, b_ex, s) =
      while_termination_point?(b_ex, body, state(
                                  (body ## catch_continue)(s)))

% proof status :−)
% do_while_TCC1 : TCC Obligation

do_while(body, b_ex) : [State −> StmtResult[State]] =
  Lambda (s : State) :
    if exists (n : nat): do_termination_point?(body, b_ex, s)(n) then
      let
        i = min[nat](do_termination_point?(body, b_ex, s))
      in (
        iterate_do(i, body, b_ex) ##
        e2s(b_ex) ## % evaluate b_ex one more time
        catch_break
      )(s)
    else
```

```
        Hang
      endif

  % proof status :-)
  do_as_while : Lemma
    do_while(body, b_ex) = (body ## catch_continue ## while(b_ex, body) ##
                              catch_break)

  do_while(max : nat, body, b_ex) : [State -> StmtResult[State]] =
    do_while(body, b_ex)

  % proof status :-)
  do_while_unroll : Lemma
    Forall (max : nat) :
      do_while(max, body, b_ex) = (body ## catch_continue ##
                                     while(max, b_ex, body) ## catch_break)

END IterationStatements1


IterationStatements2[State, ExprData : Type] : THEORY
BEGIN

  IMPORTING IterationStatements1

  s : VAR State

  b_ex: VAR [State -> ExprResult[State,bool]]
  body: VAR [State -> StmtResult[State]]

  % 6.5: A for-init-statement is either an expression-statement or a
  % simple-declaration. We have not modeled the latter (yet).
  init: VAR [State -> StmtResult[State]]
  expr: VAR [State -> ExprResult[State,ExprData]]

  % 6.5.3
  % for

  % proof status :-)
  % iterate_for_TCC1 : TCC Obligation

  % proof status :-)
  % iterate_for_TCC2 : TCC Obligation

  % unroll the loop n times (must return the state immediately before
  % evaluation of b_ex, so that we can define for_termination_point?)
  iterate_for(n : nat, b_ex, expr, body)(s) : RECURSIVE
      StmtResult[State] =
    (if n = 0 then
      skip
    else
      e2s(b_ex) ##
```

```
    body ##
    catch_continue ##
    e2s(expr) ##
    iterate_for(n−1, b_ex, expr, body)
  endif)(s)
MEASURE n

% proof status :−)
% iterate_for_as_while_TCC1 : TCC Obligation

% proof status :−)
iterate_for_as_while : Lemma
  Forall (n : nat) : n > 0 Implies
    iterate_for(n, b_ex, expr, body) =
      (e2s(b_ex) ## body ## catch_continue ##
       iterate_while(n − 1, expr ## b_ex, body) ## e2s(expr))

% proof status :−)
% for_termination_point?_TCC1 : TCC Obligation

for_termination_point?(b_ex, expr, body, s)(n : nat) : bool =
  let res = iterate_for(n, b_ex, expr, body)(s) in
    not OK?(res) or
    not OK?(b_ex(state(res))) or
    not data(b_ex(state(res)))

% proof status :−)
% termination_point_for_as_while_TCC1 : TCC Obligation

% proof status :−)
termination_point_for_as_while : Lemma
  Forall (n : nat) :
    n > 0 And
    OK?((e2s(b_ex) ## body ## catch_continue)(s))
      Implies
    for_termination_point?(b_ex, expr, body, s)(n) =
       while_termination_point?(expr ## b_ex, body,
          state((e2s(b_ex) ## body ## catch_continue)(s)))(n − 1)

% proof status :−)
% min_termination_point_for_as_while_TCC1 : TCC Obligation

% proof status :−)
% min_termination_point_for_as_while_TCC2 : TCC Obligation

% proof status :−)
% min_termination_point_for_as_while_TCC3 : TCC Obligation

% proof status :−)
min_termination_point_for_as_while : Lemma
  (Exists (n : nat):
          for_termination_point?(b_ex, expr, body, s)(n)) And
```

```
    OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) And
    Not min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 Implies
     min[nat](for_termination_point?(b_ex, expr, body, s)) =
       min[nat](while_termination_point?(expr ## b_ex, body,
         state[State]((e2s(b_ex) ## body ## catch_continue)(s)))) + 1

  % proof status :-)
  % nonzero_min_for_termination_point_TCC1 : TCC Obligation

  % proof status :-)
  nonzero_min_for_termination_point : Lemma
    (Exists (n : nat):
             for_termination_point?(b_ex, expr, body, s)(n)) And
    Not min[nat](for_termination_point?(b_ex, expr, body, s)) = 0
      Implies
    OK?(b_ex(s)) And data(b_ex(s))

  for(init, b_ex, expr, body) : [State -> StmtResult[State]] =
    (init ##
     lambda (state : State):
       if exists (n : nat):
            for_termination_point?(b_ex, expr, body, state)(n) then
         let
           i = min[nat](for_termination_point?(b_ex, expr, body, state))
         in (
           iterate_for(i, b_ex, expr, body) ##
           e2s(b_ex) ## % evaluate b_ex one more time
           catch_break
         )(state)
       else
         Hang
       endif
    )

  % proof status :-)
  for_as_while : Lemma
    for(init, b_ex, expr, body) =
      (init ##
       if_else(b_ex, body ## catch_continue ## while(expr ## b_ex, body) ##
                 catch_break, skip))

  for(max : nat, init, b_ex, expr, body) : [State -> StmtResult[State]] =
    for(init, b_ex, expr, body)

  % proof status :-)
  for_unroll : Lemma
    Forall (max : nat) :
      for(max, init, b_ex, expr, body) =
        (init ##
         if_else(b_ex, body ## catch_continue ##
                   while(max, expr ## b_ex, body) ## catch_break, skip))
```

```
END IterationStatements2

IterationStatements3[State, Ord : Type] : Theory
Begin

  Importing IterationStatements1

  b_ex : VAR [State -> ExprResult[State, bool]]
  body : VAR [State -> StmtResult[State]]

  % data invariant (P = abnormal termination, Q = normal termination)
  % termination invariant R
  P, Q, R : VAR [State -> PRED[StmtResult[State]]]

  % variant
  variant : VAR [State -> [State -> Ord]]
  < : VAR PRED[[Ord, Ord]]

  % while with embedded loop invariants

  while(b_ex, body)(P, Q, R, variant, <) : [State -> StmtResult[State]] =
    while(b_ex, body)

  do_while(body, b_ex)(P, Q, R, variant, <) : [State -> StmtResult[State]] =
    do_while(body, b_ex)

  % proof status :-)
  do_while_inv_unroll : Lemma
    do_while(body, b_ex)(P, Q, R, variant, <) =
      (body ## catch_continue ## while(b_ex, body)(P, Q, R, variant, <) ##
       catch_break)

End IterationStatements3


IterationStatements4[State, ExprData, Ord : Type] : Theory
Begin

  Importing IterationStatements2, IterationStatements3

  b_ex : VAR [State -> ExprResult[State, bool]]
  body : VAR [State -> StmtResult[State]]

  init: VAR [State -> StmtResult[State]]
  expr: VAR [State -> ExprResult[State,ExprData]]

  % data invariant (P = abnormal termination, Q = normal termination)
  % termination invariant R
  P, Q, R : VAR [State -> PRED[StmtResult[State]]]

  % variant
  variant : VAR [State -> [State -> Ord]]
```

```
  < : VAR PRED[[Ord, Ord]]

  % while with embedded loop invariants

  for(init, b_ex, expr, body)(P, Q, R, variant, <) :
      [State -> StmtResult[State]] =
    for(init, b_ex, expr, body)

  % proof status :-)
  for_inv_unroll : Lemma
    for(init, b_ex, expr, body)(P, Q, R, variant, <) =
      (init ##
       if_else(b_ex, body ## catch_continue ##
                 while(expr ## b_ex, body)(P, Q, R, variant, <) ##
                 catch_break, skip))

END IterationStatements4


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.6: Jump statements (break, continue, return, goto)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

JumpStatements1[State : Type] : THEORY
BEGIN

  IMPORTING State_Transformer

  s : VAR State
% label : VAR string

  % 6.6.1
  break : [State -> StmtResult[State]] =
    Lambda (s : State) : Break(s)

  % 6.6.2
  continue : [State -> StmtResult[State]] =
    Lambda (s : State) : Continue(s)

  % 6.6.3
  return_void : [State -> StmtResult[State]] =
    return_result

  % 6.6.4
  % goto: not modeled
% goto(label)(s) : StmtResult[State] = Goto(s, label)

END JumpStatements1
```

```
JumpStatements2[State, Data : Type] : THEORY
BEGIN

  IMPORTING Cpp_Types,
            ExpressionStatements, Abstract_Read_Write_Plain

  dt : VAR (interpreted_data_type?[Data])
  addr : VAR Address
  pm : VAR Plain_Memory[State]

  % % 6.6.3
  % MV: this is bogus as we have to potentially call the copy constructor
  % for compound objects; don't do it but generate an assignment to ret_addr
  % instead
  return(pm, dt, addr)(ex : ET[State, Data]) : [State -> StmtResult[State]] =
  (
    e2s(
      ex ## lambda (e : Data) :
      write_data(pm, dt)(addr, e)
    ) ##
    return_result
  )

END JumpStatements2


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% 6.7: Delaration statement
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DeclarationStatements1[State : Type] : THEORY
BEGIN

  IMPORTING Abstract_Read_Write_Plain, CatchAux

  fstmt : VAR [Address -> [State -> StmtResult[State]]]
  addr : Var Address
  uidt : Var Uninterpreted_data_type % to determine size

  % cf. 6.7.2, 6.6.2

  % MV >> Use cpp type instead of dt
  % This way we can allocate with proper alignment
  % MV >> do we have non-fundamental ExprResults anyway

  allocate_stack(uidt) : [State -> ExprResult[State, Address]] =
    ok_result(Mem(0)) % TODO: return proper stack address ;
                      % allocate size(uidt) bytes

  deallocate_stack(uidt, addr) : [State -> StmtResult[State]] =
```

```
    Lambda (s : State) : OK(s) % TODO: modify stack (release memory again)

  with_new_stackvar(uidt)(fstmt) : [State -> StmtResult[State]] =
  (
    allocate_stack(uidt) ## Lambda (addr : Address) :
    fstmt(addr) ##
    lift_destructor(deallocate_stack(uidt, addr))
  )

END DeclarationStatements1


DeclarationStatements2[State, Data : Type] : THEORY
BEGIN

  IMPORTING DeclarationStatements1

  fstmt : VAR [Address -> [State -> StmtResult[State]]]
  addr : Var Address
  uidt : Var Uninterpreted_data_type % to determine size
  dt : Var (interpreted_data_type?[Data])
  pm : Var Plain_Memory[State]

  with_new_returnvar(pm, dt)(fstmt) : [State -> ExprResult[State, Data]] =
  (
    allocate_stack(uidt(dt)) ## Lambda (addr : Address) :
    fstmt(addr) ##
    catch_return ##
    read_data(pm, dt)(addr)
  )

END DeclarationStatements2


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Assembler statement (this is NOT a C++ statement)
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

AssemblerStatements[State: Type] : THEORY
BEGIN

  IMPORTING SkipStatements, State_Transformer

  str : VAR string

  % We do not model assembler statements. In fact, we can't model them in
  % general, unless we know a lot more about the machine state. This
  % definition is provided only so that the semantics compiler can translate
  % assembler statements to type-correct (but unverifiable) PVS code.
```

```
  asm(str) : [State -> StmtResult[State]] =
    Lambda (s : State) : Fatal

END AssemblerStatements


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Statements: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Statements[State : Type] : THEORY
BEGIN

  IMPORTING LabeledStatements,
            % SkipStatements,
            % ExpressionStatements,
            % SelectionStatements,
            % IterationStatements1,
            % IterationStatements2,
            % IterationStatements3,
            IterationStatements4,
            JumpStatements1,
            JumpStatements2,
            % DeclarationStatements1,
            DeclarationStatements2,
            AssemblerStatements

END Statements
```

B.31 state-transformer.pvs

```
% $Id: state-transformer.pvs,v 1.61.6.1.4.2 2008/05/14 14:25:00 tews Exp $
%
% Author: Hendrik Tews
% Author: Marcus Voelp
% Author: Tjark Weber, (c) 2007 Radboud University
%
% Lifting and composition of state transformers.

Transformer_Super_Embedding_Expr[State, Data : Type] : THEORY
% map expression and statment transformers into super transformers
% utility lemmas to break up the super transformers
BEGIN
  IMPORTING Super_Embedding

  s : VAR State
  expr : VAR [State -> ExprResult[State, Data]]

  expr_2_super(expr)(s) : SuperResult[State] =
    expr_2_super_res(expr(s))
```

```
  % proof status :-)
  ok_expr_2_super : LEMMA
    OK?(expr_2_super(expr)(s)) = OK?(expr(s))

  % proof status :-)
  has_next_state_expr_2_super : LEMMA
    has_next_state(expr_2_super(expr)(s)) = has_next_state(expr(s))

  % state_expr_2_super_TCC1

  % state_expr_2_super_TCC2

  % proof status :-)
  % state_expr_2_super_TCC1 : TCC Obligation

  % proof status :-)
  % state_expr_2_super_TCC2 : TCC Obligation

  % proof status :-)
  state_expr_2_super : LEMMA
    has_next_state(expr(s)) implies
      state(expr_2_super(expr)(s)) = state(expr(s))

End Transformer_Super_Embedding_Expr

Transformer_Super_Embedding_Stmt[State : Type] : THEORY
% map expression and statment transformers into super transformers
% utility lemmas to break up the super transformers
BEGIN
  IMPORTING Super_Embedding

  s : VAR State
  stmt : VAR [State -> StmtResult[State]]

  stmt_2_super(stmt)(s) : SuperResult[State] =
    stmt_2_super_res(stmt(s))

  % proof status :-)
  ok_stmt_2_super : LEMMA
    OK?(stmt_2_super(stmt)(s)) = OK?(stmt(s))

  % proof status :-)
  has_next_state_stmt_2_super : LEMMA
    has_next_state(stmt_2_super(stmt)(s)) = has_next_state(stmt(s))

  % state_stmt_2_super_TCC1

  % state_stmt_2_super_TCC2

  % proof status :-)
  % state_stmt_2_super_TCC1 : TCC Obligation
```

```
  % proof status :-)
  % state_stmt_2_super_TCC2 : TCC Obligation

  % proof status :-)
  state_stmt_2_super : LEMMA
    has_next_state(stmt(s)) implies
      state(stmt_2_super(stmt)(s)) = state(stmt(s))

End Transformer_Super_Embedding_Stmt

Transformer_Super_Embedding : Theory
Begin
  Importing Transformer_Super_Embedding_Stmt, Transformer_Super_Embedding_Expr

End Transformer_Super_Embedding


State_Transformer_Lift_Expr[State, Data : Type] : THEORY
BEGIN

  IMPORTING Transformer_Super_Embedding

  s : VAR State

  % lift data to ExprResult
  ok_result(data : Data)(s) : ExprResult[State, Data] =
    OK(s, data)

  % proof status :-)
  ok_result_ok : LEMMA Forall(data : Data) :
    OK?(ok_result(data)(s))

  % proof status :-)
  % ok_result_data_TCC1 : TCC Obligation

  % proof status :-)
  ok_result_data : LEMMA Forall (data : Data) :
    data(ok_result(data)(s)) = data

  % proof status :-)
  ok_result_state : Lemma Forall (data : Data) :
    state(ok_result(data)(s)) = s

  % proof status :-)
  expr_2_super_ok_result : LEMMA Forall(data : Data) :
    OK?(expr_2_super(ok_result(data))(s))

  % lift a constant of type "lift[Data]" to "ExprResult[State, Data]"
  ok_lift(d : lift[Data])(s) : ExprResult[State, Data] =
    CASES d OF
      bottom : Fatal,
```

```
      up(b) : OK(s, b)
    ENDCASES

  % proof status :-)
  ok_lift_ok : LEMMA Forall(d : lift[Data]) :
    up?(d) implies ok_lift(d)(s) = OK(s, down(d))

  % lift exception to ExprResult
  exception_result(ex : Exception_type)(s) : ExprResult[State, Data] =
    Exception(ex, s)

  fatal_result(s) : ExprResult[State, Data] =
    Fatal

  % proof status :-)
  has_next_state_ok_result : Lemma Forall (data : Data) :
    has_next_state(ok_result(data)(s))

  % proof status :-)
  has_next_state_fatal_result : Lemma
    Not has_next_state(fatal_result(s))

  % proof status :-)
  expr_2_super_fatal_result : Lemma
    expr_2_super(fatal_result)(s) = Bottom

  % type abbreviations
  ET : TYPE = [State -> ExprResult[State, Data]]

END State_Transformer_Lift_Expr


State_Transformer_Lift_Stmt[State : Type] : THEORY
BEGIN

  IMPORTING Transformer_Super_Embedding

  s : VAR State

  % define lifting for statements

  % "lift" lifts a function "stmt"
  % [State -> StmtResult[State]]
  % to
  % [StmtResult[State] -> StmtResult[State]]
  %
  % If the argument is OK, then "stmt" is applied, else "stmt" is skipped. Note
  % that input and output type must be the same.

  stmt : VAR [State -> StmtResult[State]]
  sres : VAR StmtResult[State]
```

```
lift(stmt)(sres) : StmtResult[State] =
  CASES sres OF
    OK(state) : stmt(state)
    ELSE sres
  ENDCASES

% "lift_destructor" lifts a function "stmt"
% [State -> StmtResult[State]]
% to
% [StmtResult[State] -> StmtResult[State]]
%
% If the argument is OK, Break, Continue, or Return, then "stmt" is applied,
% else "stmt" is skipped. If "stmt" returns an OK state, the previous result
% constructor (Break/Continue/Return) is then used to replace the OK
% constructor. (Useful for C++ destructors, which may need to be called
% after a break/continue/return statement.) Note that input and output type
% must be the same.

% TODO: what about exceptions?

lift_destructor(stmt)(sres) : StmtResult[State] =
  CASES sres OF
    OK(state) : stmt(state),
    Break(state) :
      CASES stmt(state) OF
        OK(newstate) : Break(newstate)
        ELSE stmt(state)
      ENDCASES,
    Continue(state) :
      CASES stmt(state) OF
        OK(newstate) : Continue(newstate)
        ELSE stmt(state)
      ENDCASES,
    Return(state) :
      CASES stmt(state) OF
        OK(newstate) : Return(newstate)
        ELSE stmt(state)
      ENDCASES
    ELSE sres
ENDCASES

% switch results
switch_result(n : int)(s) : StmtResult[State] =
  Switch(s, n)

default_result(s) : StmtResult[State] =
  Default(s)

hang_result(s) : StmtResult[State] =
  Hang

return_result(s) : StmtResult[State] =
```

```
    Return(s)

  % type abbreviations
  ST : TYPE = [State -> StmtResult[State]]

END State_Transformer_Lift_Stmt


State_Transformer_Lift : THEORY
BEGIN
  IMPORTING State_Transformer_Lift_Expr, State_Transformer_Lift_Stmt

END State_Transformer_Lift


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Composition of Statements and Expressions
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The composition operator ## will be heavily overloaded.
% we artificially split the different definitions into
% several theories, such that eg show-expanded-sequent will
% tell which of the many ## has been used.

Composition_Simple_Statement_Simple_Statement[State : Type] : THEORY
BEGIN

  IMPORTING State_Transformer_Lift

  s : VAR State

  stmt_1 : VAR [State -> StmtResult[State]]
  stmt_2 : VAR [State -> StmtResult[State]]

  % Note that ## is left-associative, which is important in cases like
  % (s1 ## s2) ## catch_continue
  % vs.
  % s1 ## (s2 ## catch_continue)
  % where the treatment of a "Continue" raised in s1 depends on the
  % parentheses!

  ##(stmt_1, stmt_2)(s) : StmtResult[State] =
    lift(stmt_2)(stmt_1(s))

END Composition_Simple_Statement_Simple_Statement


Composition_Simple_Statement_Lifted_Statement[State : Type] : THEORY
BEGIN
```

```
  IMPORTING StmtResult

  s : VAR State
  stmt_1 : VAR [State -> StmtResult[State]]
  lift_stmt_2 : VAR [StmtResult[State] -> StmtResult[State]]

  % Example: whilebody ## catch_continue
  ##(stmt_1, lift_stmt_2)(s) : StmtResult[State] =
    lift_stmt_2(stmt_1(s))

END Composition_Simple_Statement_Lifted_Statement


Composition_Lifted_Statement_Simple_Statement[State : Type] : THEORY
BEGIN

  IMPORTING State_Transformer_Lift

  sres : VAR StmtResult[State]
  lift_stmt_1 : VAR [StmtResult[State] -> StmtResult[State]]
  stmt_2 : VAR [State -> StmtResult[State]]

  % Example: (case 1) ## (if ...)
  ##(lift_stmt_1, stmt_2)(sres) : StmtResult[State] =
    lift(stmt_2)(lift_stmt_1(sres))

END Composition_Lifted_Statement_Simple_Statement


Composition_Lifted_Statement_Lifted_Statement[State : Type] : THEORY
BEGIN

  IMPORTING StmtResult

  sres : VAR StmtResult[State]
  lift_stmt_1 : VAR [StmtResult[State] -> StmtResult[State]]
  lift_stmt_2 : VAR [StmtResult[State] -> StmtResult[State]]

  % Example: ((case 1) ## ... ) ## (case 2)
  ##(lift_stmt_1, lift_stmt_2)(sres) : StmtResult[State] =
    lift_stmt_2(lift_stmt_1(sres))

END Composition_Lifted_Statement_Lifted_Statement


Composition_Simple_Expression_Data_Expression[State, Data1, Data2 : Type] :
    THEORY
BEGIN

  IMPORTING Transformer_Super_Embedding

  s : VAR State
```

```
  expr : VAR [State -> ExprResult[State, Data1]]
  fexpr : VAR [Data1 -> [State -> ExprResult[State, Data2]]]

  % expression ## (lambda data: expression)

  ##(expr, fexpr)(s) : ExprResult[State, Data2] =
    CASES expr(s) OF
      OK(state, data) : fexpr(data)(state),
      Exception(ex_type, state) : Exception(ex_type, state),
      Fatal : Fatal,
      Hang : Hang
      % TODO: other expression results?
    ENDCASES

  % super_comp_expr_expr_TCC1

  % super_comp_expr_expr_TCC1

  % proof status :-)
  % super_comp_expr_expr_TCC1 : TCC Obligation

  % proof status :-)
  % super_comp_expr_expr_TCC2 : TCC Obligation

  % proof status :-)
  super_comp_expr_expr : Lemma
    expr_2_super(expr ## fexpr)(s) =
      CASES expr_2_super(expr)(s) OF
        OK(state) : expr_2_super(fexpr(data(expr(s))))
                                              (state(expr_2_super(expr)(s)))
        ELSE expr_2_super(expr)(s)
      ENDCASES

  % deprecated now, but kept for compatibility
  eval_if_ok(expr, fexpr) : MACRO [State -> ExprResult[State, Data2]] =
    (expr ## fexpr)

END Composition_Simple_Expression_Data_Expression


Composition_Simple_Expression_Simple_Expression[State, Data1, Data2 : Type] :
    THEORY
BEGIN

  IMPORTING Composition_Simple_Expression_Data_Expression

  s : VAR State
  expr1 : VAR [State -> ExprResult[State, Data1]]
  expr2 : VAR [State -> ExprResult[State, Data2]]

  % expression ## expression (forgetful composition)
```

```
##(expr1, expr2)(s) : ExprResult[State, Data2] =
  (expr1 ## (lambda (d : Data1): expr2))(s)

% proof status :-)
comp_expr_forget_expr : LEMMA
  (expr1 ## expr2)(s) =
    CASES expr1(s) OF
      OK(state, data) : expr2(state),
      Exception(ex_type, state) : Exception(ex_type, state),
      Fatal : Fatal,
      Hang : Hang
      % TODO: other expression results?
    ENDCASES

% proof status :-)
super_comp_expr_forget_expr : LEMMA
  expr_2_super(expr1 ## expr2)(s) =
    CASES expr_2_super(expr1)(s) OF
      OK(state) : expr_2_super(expr2)(state)
      ELSE expr_2_super(expr1)(s)
    ENDCASES

% proof status :-)
% comp_simple_expr_simple_expr_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr : Lemma
  OK?(expr1(s)) Implies
    expr2(state(expr1(s))) = (expr1 ## expr2)(s)

% proof status :-)
% comp_simple_expr_simple_expr_ok_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr_ok : Lemma
  OK?((expr1 ## expr2)(s)) Implies
    OK?(expr2(state(expr1(s))))

% proof status :-)
% comp_simple_expr_simple_expr_data_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr_data : Lemma
  OK?((expr1 ## expr2)(s)) Implies
    data(expr2(state(expr1(s)))) = data((expr1 ## expr2)(s))

% proof status :-)
% comp_simple_expr_simple_expr_state_TCC1 : TCC Obligation

% proof status :-)
comp_simple_expr_simple_expr_state : Lemma
  OK?((expr1 ## expr2)(s)) Implies
```

```
    state(expr2(state(expr1(s)))) = state((expr1 ## expr2)(s))

END Composition_Simple_Expression_Simple_Expression


Composition_Simple_Expression_Data_Statement[State, Data : Type] :
  THEORY
BEGIN

  IMPORTING ExprResult, StmtResult

  s : VAR State
  expr : VAR [State -> ExprResult[State, Data]]
  fstmt : VAR [Data -> [State -> StmtResult[State]]]

  % expression ## (lambda data: statement)

  ##(expr, fstmt)(s) : StmtResult[State] =
    CASES expr(s) OF
      OK(state, data) : fstmt(data)(state),
      Exception(ex_type, state) : Exception(ex_type, state),
      Fatal : Fatal,
      Hang : Hang
      % TODO: other expression results?
    ENDCASES

END Composition_Simple_Expression_Data_Statement


Composition_Simple_Statement_Simple_Expression[State, Data : Type] :
  THEORY
BEGIN

  IMPORTING StmtResult, ExprResult

  s : VAR State
  stmt : VAR [State -> StmtResult[State]]
  expr : VAR [State -> ExprResult[State, Data]]

  % define lifting for expressions

  sres : VAR StmtResult[State]

  lift(expr)(sres) : ExprResult[State, Data] =
    CASES sres OF
      OK(state) : expr(state),
      Hang : Hang,
      Exception(ex_type, state) : Exception(ex_type, state)
      ELSE
        Fatal
    ENDCASES;
```

```
  % Example: stmt ## read_data(...)
  ##(stmt, expr)(s) : ExprResult[State, Data] =
    lift(expr)(stmt(s))

  % useful for Hoare logic, to append expressions in the postcondition
  % (e.g. read_data(...)) to the program

  % comp_simple_stmt_simple_expr_TCC1

  % proof status :-)
  % comp_simple_stmt_simple_expr_TCC1 : TCC Obligation

  % proof status :-)
  comp_simple_stmt_simple_expr : LEMMA
      OK?(stmt(s)) implies
        expr(state(stmt(s))) = (stmt ## expr)(s)

  % proof status :-)
  % comp_simple_stmt_simple_expr_ok_TCC1 : TCC Obligation

  % proof status :-)
  comp_simple_stmt_simple_expr_ok : Lemma
    OK?((stmt ## expr)(s)) Implies
      OK?(expr(state(stmt(s))))

  % proof status :-)
  % comp_simple_stmt_simple_expr_data_TCC1 : TCC Obligation

  % proof status :-)
  comp_simple_stmt_simple_expr_data : Lemma
    OK?((stmt ## expr)(s)) Implies
      data(expr(state(stmt(s)))) = data((stmt ## expr)(s))

  % proof status :-)
  % comp_simple_stmt_simple_expr_state_TCC1 : TCC Obligation

  % proof status :-)
  comp_simple_stmt_simple_expr_state : Lemma
    OK?((stmt ## expr)(s)) Implies
      state(expr(state(stmt(s)))) = state((stmt ## expr)(s))

END Composition_Simple_Statement_Simple_Expression


Composition_Simple_Statement_Lifted_Expression[State, Data : Type] :
    THEORY
BEGIN

  IMPORTING StmtResult, ExprResult

  s : VAR State
  stmt : VAR [State -> StmtResult[State]]
```

```
  lift_expr : VAR [StmtResult[State] -> ExprResult[State, Data]]

  % Example: function_body ## catch_return
  ##(stmt, lift_expr)(s) : ExprResult[State, Data] =
    lift_expr(stmt(s))

END Composition_Simple_Statement_Lifted_Expression


Composition_Lifted_Statement_Simple_Expression[State, Data : Type] :
    THEORY
BEGIN

  IMPORTING Composition_Simple_Statement_Simple_Expression

  sres : VAR StmtResult[State]
  lift_stmt_1 : VAR [StmtResult[State] -> StmtResult[State]]
  expr_2 : VAR [State -> ExprResult[State, Data]]

  ##(lift_stmt_1, expr_2)(sres) : ExprResult[State, Data] =
    lift(expr_2)(lift_stmt_1(sres))

END Composition_Lifted_Statement_Simple_Expression


Composition_Lifted_Statement_Lifted_Expression[State, Data : Type] :
    THEORY
BEGIN

  IMPORTING Composition_Simple_Statement_Simple_Expression

  sres : VAR StmtResult[State]
  lift_stmt : VAR [StmtResult[State] -> StmtResult[State]]
  lift_expr : VAR [StmtResult[State] -> ExprResult[State, Data]]

  ##(lift_stmt, lift_expr)(sres) : ExprResult[State, Data] =
    lift_expr(lift_stmt(sres))

END Composition_Lifted_Statement_Lifted_Expression


Statement_Composition_Rewrites[State: Type] : THEORY
BEGIN

  IMPORTING Composition_Simple_Statement_Simple_Statement,
            Composition_Simple_Statement_Lifted_Statement,
            Composition_Lifted_Statement_Simple_Statement,
            Composition_Lifted_Statement_Lifted_Statement

  s : VAR State
  sres : VAR StmtResult[State]
  stmt_1 : VAR [State -> StmtResult[State]]
```

```
stmt_2 : Var [State -> StmtResult[State]]
stmt_3 : Var [State -> StmtResult[State]]
lift_stmt_1 : VAR [StmtResult[State] -> StmtResult[State]]
lift_stmt_2 : VAR [StmtResult[State] -> StmtResult[State]]
lift_stmt_3 : VAR [StmtResult[State] -> StmtResult[State]]

% For symbolic evaluation, we want a composition operator that associates
% to the right: e.g.
% ((s1 ## s2) ## catch_continue)(s)
% must be rewritten to
% (s1 ## (lift(s2) ## catch_continue))(s)
% before evaluation can be performed, starting with s1.
%
% With 3 statements, each one either lifted or not, there are 8 cases to
% consider.
%
% Essentially: when stmt_2 is not lifted, but stmt_3 is, then stmt_2 must be
% lifted.

% proof status :-)
composition_assoc_rewrite_1 : LEMMA
  (stmt_1 ## (stmt_2 ## stmt_3)) =
  ((stmt_1 ## stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_2 : LEMMA
  (stmt_1 ## (lift(stmt_2) ## lift_stmt_3)) =
  ((stmt_1 ## stmt_2) ## lift_stmt_3)

% proof status :-)
composition_assoc_rewrite_3 : LEMMA
  (stmt_1 ## (lift_stmt_2 ## stmt_3)) =
  ((stmt_1 ## lift_stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_4 : LEMMA
  (stmt_1 ## (lift_stmt_2 ## lift_stmt_3)) =
  ((stmt_1 ## lift_stmt_2) ## lift_stmt_3)

% proof status :-)
composition_assoc_rewrite_5 : LEMMA
  (lift_stmt_1 ## (stmt_2 ## stmt_3)) =
  ((lift_stmt_1 ## stmt_2) ## stmt_3)

% proof status :-)
composition_assoc_rewrite_6 : LEMMA
  equalities[[StmtResult[State] -> StmtResult[State]]].= (
    (lift_stmt_1 ## (lift(stmt_2) ## lift_stmt_3)),
    ((lift_stmt_1 ## stmt_2) ## lift_stmt_3))

% proof status :-)
composition_assoc_rewrite_7 : LEMMA
```

```
    (lift_stmt_1 ## (lift_stmt_2 ## stmt_3)) =
    ((lift_stmt_1 ## lift_stmt_2) ## stmt_3)

  % proof status :-)
  composition_assoc_rewrite_8 : LEMMA
    equalities[[StmtResult[State] -> StmtResult[State]]].= (
      (lift_stmt_1 ## (lift_stmt_2 ## lift_stmt_3)),
      ((lift_stmt_1 ## lift_stmt_2) ## lift_stmt_3))

END Statement_Composition_Rewrites


Expression_Composition_Rewrites[State, Data1, Data2 : Type] : THEORY
BEGIN

  IMPORTING Composition_Simple_Expression_Simple_Expression

  s : VAR State
  expr : VAR [State -> ExprResult[State, Data1]]
  fexpr : VAR [Data1 -> [State -> ExprResult[State, Data2]]]

  % proof status :-)
  comp_eval_if_ok_fexpr : LEMMA
      OK?(expr(s)) implies
        (expr ## fexpr)(s) = (expr ## fexpr(data(expr(s))))(s)

END Expression_Composition_Rewrites


Ok_Result_Rewrite[State, Data1, Data2 : Type] : Theory
Begin

  Importing State_Transformer_Lift,
            Composition_Simple_Expression_Data_Statement,
            Expression_Composition_Rewrites,
            Composition_Lifted_Statement_Simple_Expression

  s : Var State
  data : Var Data1
  data2 : Var Data2
  q : Var [State -> ExprResult[State, Data2]]
  states : Var PRED[State]



  % ok_result
  %-------------
  % proof status :-)
  ok_result_q_ok : Lemma
    OK?[State, Data1]((q ## ok_result(data))(s)) = OK?(q(s))

  % proof status :-)
```

```
  % ok_result_q_state_TCC1 : TCC Obligation

  % proof status :-)
  % ok_result_q_state_TCC2 : TCC Obligation

  % proof status :-)
  ok_result_q_state : Lemma
    OK?(q(s)) Implies
      state[State, Data1]((q ## ok_result(data))(s)) = state(q(s))

  % proof status :-)
  % ok_result_q_data_TCC1 : TCC Obligation

  % proof status :-)
  ok_result_q_data : Lemma
    OK?(q(s)) Implies
      data((q ## ok_result(data))(s)) = data

  % proof status :-)
  ok_result_elemination_1 : Lemma
    (ok_result[State, Data1](data) ## q) = q

End Ok_Result_Rewrite


Transformer_Invariant[State : Type] : Theory
Begin
  IMPORTING Super_Result_Util

  transformers : Var PRED[[State -> SuperResult[State]]]
  states : Var PRED[State]


  % proof status :-)
  % result_pred_TCC1 : TCC Obligation

  result_pred(states) : PRED[SuperResult[State]] =
    Lambda(res : SuperResult[State]) :
      has_next_state(res) Implies states(state(res))

  % invariants for state transformers
  transformer_invariant?(states, transformers) : bool =
    Forall(s : State, q : [State -> SuperResult[State]]) :
      states(s) AND transformers(q) IMPLIES
        result_pred(states)(q(s))

  % proof status :-)
  % super_transformer_invariant_next_ok_TCC1 : TCC Obligation

  % invariant lemma
  % proof status :-)
  super_transformer_invariant_next_ok : Lemma
```

```
    Forall(s : State, q : [State -> SuperResult[State]]) :
      transformer_invariant?(states, transformers) AND
      states(s) AND transformers(q) AND
      has_next_state(q(s))
      IMPLIES
        states(state(q(s)))

  % Monotonicity
  % proof status :-)
  transformer_invariant_mono_transformers : Lemma
    Forall(transformers_1, transformers_2 :
                                          PRED[[State -> SuperResult[State]]]) :
      subset?(transformers_1, transformers_2) AND
      transformer_invariant?(states, transformers_2)
        IMPLIES
          transformer_invariant?(states, transformers_1)

  % union of transformers
  % proof status :-)
  transformer_invariant_union_transformers : Lemma
    Forall(transformers_1, transformers_2 :
                                          PRED[[State -> SuperResult[State]]]) :
      transformer_invariant?(states, transformers_1) AND
      transformer_invariant?(states, transformers_2)
        IMPLIES
          transformer_invariant?(states, union(transformers_1, transformers_2))


  % the full set is always an invariant
  % proof status :-)
  transformer_invariant_truth : Lemma
    transformer_invariant?(fullset[State], transformers)

  % proof status :-)
  transformer_invariant_all_transformers : Lemma
    (Forall (q : [State -> SuperResult[State]]) : transformers(q) Implies
       transformer_invariant?(states, singleton(q)))
    Implies
      transformer_invariant?(states, transformers)

  % all transformers return OK
  transformers_ok?(states, transformers) : bool =
    Forall(s : State, q : [State -> SuperResult[State]]) :
      states(s) AND transformers(q)
        IMPLIES
          OK?(q(s))

  % access lemma
  % proof status :-)
  super_transformers_ok_ok : Lemma
    Forall(s : State, q : [State -> SuperResult[State]]) :
      transformers_ok?(states, transformers) AND
```

```
      states(s) AND transformers(q) IMPLIES
        OK?(q(s))

  % Monotonicity
  % proof status :-)
  transformers_ok_mono_transformers : Lemma
    Forall(transformers_1, transformers_2 :
                                        PRED[[State -> SuperResult[State]]]) :
      subset?(transformers_1, transformers_2) AND
      transformers_ok?(states, transformers_2)
        IMPLIES
          transformers_ok?(states, transformers_1)

  % proof status :-)
  transformers_ok_mono_states : Lemma
    Forall(states_1, states_2 : PRED[State]) :
      subset?(states_1, states_2) AND
      transformers_ok?(states_2, transformers)
        IMPLIES
          transformers_ok?(states_1, transformers)

  % lemma for union
  % proof status :-)
  transformers_ok_union_transformers : Lemma
    Forall(transformers_1, transformers_2 :
                                        PRED[[State -> SuperResult[State]]]) :
      transformers_ok?(states, transformers_1) AND
      transformers_ok?(states, transformers_2)
        IMPLIES
          transformers_ok?(states, union(transformers_1, transformers_2))

  % proof status :-)
  transformers_ok_all_transformers : Lemma
    (Forall (q : [State -> SuperResult[State]]) : transformers(q) Implies
       transformers_ok?(states, singleton(q)))
    Implies
      transformers_ok?(states, transformers)


End Transformer_Invariant


Transformer_Invariant_2[State, Data : Type] : Theory
% express some of above lemmas for non-super transformers
Begin
  Importing Transformer_Invariant, Transformer_Super_Embedding

  transformers : Var PRED[[State -> SuperResult[State]]]
  states : Var PRED[State]

  % proof status :-)
  % expr_transformer_invariant_next_ok_TCC1 : TCC Obligation
```

```
  % proof status :-)
  expr_transformer_invariant_next_ok : Lemma
    Forall(s : State, q : [State -> ExprResult[State, Data]]) :
      transformer_invariant?(states, transformers) AND
      states(s) AND transformers(expr_2_super(q)) AND
      has_next_state(q(s))
      IMPLIES
        states(state(q(s)))


  % proof status :-)
  expr_transformers_ok_ok : Lemma
    Forall(s : State, q : [State -> ExprResult[State, Data]]) :
      transformers_ok?(states, transformers) AND
      states(s) AND transformers(expr_2_super(q)) IMPLIES
        OK?(q(s))

  % proof status :-)
  % expr_result_pred_next_state_TCC1 : TCC Obligation

  % proof status :-)
  expr_result_pred_next_state : Lemma
    Forall(s : State, q : [State -> ExprResult[State, Data]]) :
      has_next_state(q(s)) And
      result_pred(states)(expr_2_super(q)(s)) Implies
        states(state(q(s)))

End Transformer_Invariant_2


Transformer_Invariant_3[State, Data1, Data2 : Type] : Theory
Begin
  Importing Transformer_Invariant_2, Expression_Composition_Rewrites

  states : Var PRED[State]

  expr1 : Var [State -> ExprResult[State, Data1]]
  expr2 : Var [State -> ExprResult[State, Data2]]
  fexpr : Var [Data1 -> [State -> ExprResult[State, Data2]]]

  % proof status :-)
  fexpr_composition_transformers_ok : Lemma
    Forall (P : PRED[Data1]) :
      transformers_ok?(states, singleton(expr_2_super(expr1))) And
      transformer_invariant?(states, singleton(expr_2_super(expr1))) And
      (Forall (s : (states)) : OK?(expr1(s)) Implies P(data(expr1(s)))) And
      (Forall (d : (P)) : transformers_ok?(states, singleton(expr_2_super(fexpr(d)))))
    Implies
      transformers_ok?(states, singleton(expr_2_super(expr1 ## fexpr)))

  % proof status :-)
```

```
  expr_composition_transformers_ok : Lemma
    transformers_ok?(states, singleton(expr_2_super(expr1))) And
    transformer_invariant?(states, singleton(expr_2_super(expr1))) And
    transformers_ok?(states, singleton(expr_2_super(expr2)))
  Implies
    transformers_ok?(states, singleton(expr_2_super(expr1 ## expr2)))

  % proof status :-)
  fexpr_composition_transformer_invariant : Lemma
    Forall (P : PRED[Data1]) :
      transformer_invariant?(states, singleton(expr_2_super(expr1))) And
      (Forall (s : (states)) : OK?(expr1(s)) Implies P(data(expr1(s)))) And
      (Forall (d : (P)) : transformer_invariant?(states, singleton(expr_2_super(fexpr(d)))))
    Implies
      transformer_invariant?(states, singleton(expr_2_super(expr1 ## fexpr)))

  % proof status :-)
  expr_composition_transformer_invariant : Lemma
    transformer_invariant?(states, singleton(expr_2_super(expr1))) And
    transformer_invariant?(states, singleton(expr_2_super(expr2)))
  Implies
    transformer_invariant?(states, singleton(expr_2_super(expr1 ## expr2)))

End Transformer_Invariant_3


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% State_Transformer: imports everything
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

State_Transformer : Theory
BEGIN

  IMPORTING
            % Transformer_Super_Embedding,
            % State_Transformer_Lift,
            % Composition_Simple_Statement_Simple_Statement,
            % Composition_Simple_Statement_Lifted_Statement,
            % Composition_Lifted_Statement_Simple_Statement,
            % Composition_Lifted_Statement_Lifted_Statement,
            % Composition_Simple_Expression_Data_Expression,
            % Composition_Simple_Expression_Simple_Expression,
            % Composition_Simple_Expression_Data_Statement,
            % Composition_Simple_Statement_Simple_Expression,
            Composition_Simple_Statement_Lifted_Expression,
            Composition_Lifted_Statement_Lifted_Expression,
            Statement_Composition_Rewrites,
            % Expression_Composition_Rewrites,
            Ok_Result_Rewrite,
% Make_Transformer,
```

```
        % Transformer_Invariant,
        %Transformer_Invariant_2,
        Transformer_Invariant_3

END State_Transformer
```

B.32 types.pvs

```
% $Id: types.pvs,v 1.16.4.2.2.4.2.4 2008/05/14 14:25:01 tews Exp $
%
% Author: Tjark Weber; Hendrik Tews; Marcus Völp et al.
% (c) 2007 Radboud University
%
% A formalization of C++ types.

% Section numbers in comments throughout this file refer to
% "Programming languages − C++", ISO/IEC 14882:1998(E)


% Open Issues / To Do
%====================
% Open issue: can we reap benefit from this?
% 3.9.1 pt 7 representation of value of integral through pure binary system
% integral value = Sum (0 <= p <(?=) n) b[p] * 2^p with bits b[p]
%
% Can we conclude from this that the max_value_bits are the bits in the
% value representation, or more precisely that:
% [bits in object representation] >= [bits in value representation]
% size(uidt(unsigned)) * bits_per_byte >= max_value_bits(unsigned)

% Open issue:
% The underlying data model with its to_byte/from_byte encoding functions
% does not really allow to model individual bits which are in a type's
% object representation, but NOT in its value representation.

% Open issue: const, volatile
% Is it wise to formalise const, volatile as Cpp_Type; there seems to be
% no easy way to define that expressions on cv(t) behave like expressions
% on t. Should we use typ(c, v : bool) instead? Note that this implies
% dt(t) = dt(volatile(t)) and thus that hidden bits cannot indicate a
% volatile access.

% MV: To do:
% − classes are not CV qualified by their members however a cv class
% behave like all nonstatic members being cv qualified ; use the semantics
% compiler to generate appropriate member types
% − cv(array) = array(cv)
% − cv void only as function return parameter
% − cv function affects cv of this => use semantics compiler to type this

% => define dt(*) only for interpreted types with domain
% Number x Address x Semantics Pointer
```

```
% -> generic handling of dt
% -> generic behaviour of const, volatile
% -> Define CV_Arithmetric
% -> dt_const(t) = dt(t)
% -> dt_volatile(t) = dt(t) except for hidden bits

% Not modelled (yet):
% - Pointer-to-function
% - const/volatile specifiers
% - float, double, long double
% - Conversion between integral types and bitsets
% - Conversion between integral types and enumeration types

% - wird kein every mehr für Datentypen definiert?
% - dt_exists TCC durch instanziierung mit konkretem Modell beweisen
% 3.9.1.1: "In any particular implementation, a plain char object can
% take on either the same values as a signed char or an
% unsigned char; which one is implementation-defined." -- We
% take this to also mean that the value representation is the
% same for char and (either) signed char or unsigned char. (Why?)

% Todo:
% - should we encode arg_type as list[Cpp_Type]
% or better as members, below[members] -> CppType
% - variable arguments
% - semantic compiler generates var_args object to contain arguments
% - var_args object is passed to function
% - default values
% - semantics compiler initialises all uninitialised values with defaults.



% See abstract_data.pvs for the underlying data type model.
%
% Naming conventions:
% - theories are named "Cpp_<type>",
% - value ranges are named "range(<type>)",
% - data types are named "dt_<type>",
% where <type> is one of
% - uchar, schar, char,
% - short, int, long, longlong,
% - ushort, uint, ulong, ulonglong,
% - wchar_t,
% - bool,
% - void,
% - pointer,
% - reference.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Fundamental Types
%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Deviations from the C++ standard:
%
% The underlying data model with its to_byte/from_byte encoding functions
% does not really allow to model individual bits which are in a type's
% object representation, but NOT in its value representation.

% 3.9.1 pt 1 Unsigned Char
%============================
Cpp_uchar : Theory
Begin

  Importing finite_sets@finite_sets_minmax[nat, <=], More_More_List_Props

  % Alignment
  %------------
  align_uchar : posnat

  % proof status :-)
  % range_uchar_TCC1 : TCC Obligation

  % Range of values
  %------------------
  range_uchar : non_empty_finite_set[nat]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_uchar)]

  dt_uchar_exists : Axiom
    Exists (dt : (pod_data_type?[(range_uchar)])) : true

  % proof status :-)
  % dt_uchar_TCC1 : TCC Obligation

  dt_uchar : (pod_data_type?[(range_uchar)])

  % 5.3.3 pt 1 "The size_of operator yields the number of bytes in the object
  % representation of its operand."
  % "size_of(unsigned char) [is] 1"

  dt_uchar_size : Axiom
    size(uidt(dt_uchar)) = 1

  % 3.9.1 pt 1: "For unsigned character types, all possible bit patterns of the
  % value representation represent numbers."
  % (TODO: This seems to follow from the arithmetic-mod-2^n requirement
  % (3.9.1 pt 4) already -- maybe we could prove it?!)
  dt_uchar_valid : Axiom
    Forall (l : list[Byte], a : Address):
      length(l) = size(uidt(dt_uchar)) Implies valid?(uidt(dt_uchar))(l, a)
```

```
  % Bits in the value representation
  %-----------------------------------
  max_value_bits_uchar : posnat

  % proof status :-)
  % value_bitmask_uchar_TCC1 : TCC Obligation

  % a bitmask in which all bits are set that are affected by a bit
  % operation on this type
  value_bitmask_uchar : {l : list[Byte] | length(l) = size(uidt(dt_uchar))}

  % C Standard, 5.2.4.2.1: UCHAR_MAX >= 255 (2^8 - 1)
  % 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
  % modulo 2^n where n is the number of bits in the value
  % representation"
  min_max_value_bits_uchar : Axiom
    max_value_bits_uchar >= 8

  % 3.9 p4: "For POD types, the value representation is the set of bits in
  % the object representation that determines a value [...]"
  % 3.9.1 pt1: "For character types, all bits of the object representation
  % participate in the value representation."
  dt_uchar_bits : Axiom
    max_value_bits_uchar = size(uidt(dt_uchar)) * bits_per_byte

  % max_value(uchar) >= 255
  max_uchar : Axiom
    max(range_uchar) >= (2^max_value_bits_uchar) - 1

  % 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
  % modulo 2^n where n is the number of bits in the value
  % representation"
  binary_range_uchar : Axiom
    Forall (n : nat) :
      range_uchar(n) IFF n <= max(range_uchar)

  % => max_value_bits(uchar) >= min_bits_per_byte
  % => max_value(uchar) = 2^bits_per_byte - 1

End Cpp_uchar

% 3.9.1 pt 1 Signed Char
%=======================
Cpp_schar : Theory
Begin

  Importing Cpp_uchar, finite_sets@finite_sets_minmax[int, <=]


  % Alignment
  %----------
```

```
  % 3.9.1 pt 1 alignment of schar = alignment of uchar
  align_schar : posnat = align_uchar

  % proof status :-)
  % range_schar_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_schar : non_empty_finite_set[int]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_schar)]

  dt_schar_exists : Axiom
    Exists (dt : (pod_data_type?[(range_schar)])) : true

  % proof status :-)
  % dt_schar_TCC1 : TCC Obligation

  dt_schar : (pod_data_type?[(range_schar)])

  % 3.9.1 pt 1 size_of(schar) = size_of(uchar)
  dt_schar_size : Axiom
    size(uidt(dt_schar)) = size(uidt(dt_uchar))

  % 3.9.1 pt 3 "The range of nonnegative values of a signed integer type
  % is the range of the corresponding unsigned integer type,
  % and the value representation of each corresponding
  % signed / unsigned type shall be the same."
  schar_nonneg_is_uchar : Axiom
    Forall (i : (range_schar)) : i >= 0 Implies range_uchar(i)

  % proof status :-)
  % schar_value_eq_uchar_value_TCC1 : TCC Obligation

  schar_value_eq_uchar_value : Axiom
    Forall (i : (range_schar), a : Address) : i >= 0 Implies
      to_byte(dt_schar)(i, a) = to_byte(dt_uchar)(i, a)

  max_value_bits_schar : nat

  % proof status :-)
  % value_bitmask_schar_TCC1 : TCC Obligation

  value_bitmask_schar : {l : list[Byte] | length(l) = size(uidt(dt_schar))}

  % 3.9 pt 4: "For POD types, the value representation is the set of bits in
  % the object representation that determines a value [...]"
  % 3.9.1 pt 1: "For character types, all bits of the object representation
  % participate in the value representation."
  % The highest order bit may however encode the sign of the value.
```

```
  % (footnote 49 3.9.1 pt 7)
  dt_schar_bits : Axiom
    max_value_bits_schar >= size(uidt(dt_schar)) * bits_per_byte - 1

  % C Standard, 5.2.4.2.1: SCHAR_MIN <= -127 (-(2^7 - 1))
  % C Standard, 5.2.4.2.1: SCHAR_MAX >= 127 (2^7 - 1)
  min_schar : Axiom
    min(range_schar) <= -((2 ^ max_value_bits_schar) - 1)

  max_schar : Axiom
    max(range_schar) >= (2 ^ max_value_bits_schar) - 1

  schar_binary : Axiom
    Forall (i : int) :
      range_schar(i) IFF (min(range_schar) <= i And i <= max(range_schar))

  % => min_value(schar) <= -127
  % => max_value(schar) >= 127
  % => max_value_bits(schar) >= min_bits_per_byte - 1

End Cpp_schar


% 3.9.1 pt 1 Char
%=================
Cpp_char : Theory
Begin

  Importing Cpp_schar

  % Alignment
  %------------
  % 3.9.1 pt 1 alignment of char = alignment of uchar
  align_char : posnat = align_uchar

  % Range of values
  %-----------------
  range_char : non_empty_finite_set[int]

  % 3.9.1.1: "In any particular implementation, a plain char object can
  % take on either the same values as a signed char or an
  % unsigned char; which one is implementation-defined."
  char_is_uchar_or_schar : Axiom
    range_char = range_schar OR range_char = range_uchar

  % (Un)interpreted Datatype
  %-----------------------
  Importing Interpreted_Data[(range_char)]

  dt_char_exists : Axiom
    Exists (dt : (pod_data_type?[(range_char)])) : true
```

```
  % proof status :-)
  % dt_char_TCC1 : TCC Obligation

  dt_char : (pod_data_type?[(range_char)])

  % 3.9.1 pt 1 size_of(schar) = size_of(uchar)
  dt_char_size : Axiom
    size(uidt(dt_char)) = size(uidt(dt_uchar))

  max_value_bits_char : nat

  % proof status :-)
  % value_bitmask_char_TCC1 : TCC Obligation

  value_bitmask_char : {l : list[Byte] | length(l) = size(uidt(dt_char))}

  % 3.9 pt 4: "For POD types, the value representation is the set of bits in
  % the object representation that determines a value [...]"
  % 3.9.1 pt 1: "For character types, all bits of the object representation
  % participate in the value representation."
  % The highest order bit may however encode the sign of the value.
  % (footnote 49 3.9.1 pt 7)
  dt_char_bits : Axiom
    max_value_bits_char >= size(uidt(dt_char)) * bits_per_byte - 1

  % proof status :-)
  min_char : Lemma
    min(range_char) <= 0

  % proof status :-)
  max_char : Lemma
    max(range_char) >= 127

  % proof status :-)
  char_binary : Lemma
    Forall (i : int) :
      range_char(i) IFF (min(range_char) <= i And i <= max(range_char))

End Cpp_char


% 3.9.1 pt 2 Signed Short Int
%-------------------------
Cpp_short : Theory
Begin

  Importing Cpp_schar, finite_sets@finite_sets_minmax[int, <=]

  % Alignment
  %-----------
  align_short : posnat
```

```
  % proof status :-)
  % range_short_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_short : non_empty_finite_set[int]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_short)]

  dt_short_exists : Axiom
    Exists (dt : (pod_data_type?[(range_short)])) : true

  % proof status :-)
  % dt_short_TCC1 : TCC Obligation

  dt_short : (pod_data_type?[(range_short)])

  % 3.9.1.2: "There are four signed integer types: signed char,
  % short int, int, and long int. In this list, each type
  % provides at least as much storage as those preceding it
  % in the list." -- We take this to mean that both the value
  % range and the size of the object representation must not
  % be smaller.
  dt_short_size : Axiom
    size(uidt(dt_short)) >= size(uidt(dt_schar))

  short_value_supset_schar_value : Axiom
    Forall (i : (range_schar)) : range_short(i)

  max_value_bits_short : nat

  % proof status :-)
  % value_bitmask_short_TCC1 : TCC Obligation

  value_bitmask_short : {l : list[Byte] | length(l) = size(uidt(dt_short))}

  % C Standard, 5.2.4.2.1: SHRT_MIN <= -32767 (-(2^15 - 1))
  min_short : Axiom
    min(range_short) <= -32767

  % C Standard, 5.2.4.2.1: SHRT_MAX >= 32767 (2^15 - 1)
  max_short : Axiom
    max(range_short) >= 32767

  short_binary : Axiom
    FORALL (i: int):
      range_short(i) IFF (min(range_short) <= i And i <= max(range_short))

End Cpp_short
```

```
% 3.9.1 pt 2 Signed Int
%-----------------------------
Cpp_int : Theory
Begin

  Importing Cpp_short, finite_sets@finite_sets_minmax[int, <=]

  % Alignment
  %------------
  align_int : posnat

  % proof status :-)
  % range_int_TCC1 : TCC Obligation

  % Range of values
  %-------------------
  range_int : non_empty_finite_set[int]

  % (Un)interpreted Datatype
  %--------------------------
  Importing Interpreted_Data[(range_int)]

  dt_int_exists : Axiom
    Exists (dt : (pod_data_type?[(range_int)])) : true

  % proof status :-)
  % dt_int_TCC1 : TCC Obligation

  dt_int : (pod_data_type?[(range_int)])

  % 3.9.1.2: "There are four signed integer types: signed char,
  % short int, int, and long int. In this list, each type
  % provides at least as much storage as those preceding it
  % in the list." -- We take this to mean that both the value
  % range and the size of the object representation must not
  % be smaller.
  dt_int_size : Axiom
    size(uidt(dt_int)) >= size(uidt(dt_short))

  int_value_supset_short_value : Axiom
    Forall (i : (range_short)) : range_int(i)

  max_value_bits_int : nat

  % proof status :-)
  % value_bitmask_int_TCC1 : TCC Obligation

  value_bitmask_int : {l : list[Byte] | length(l) = size(uidt(dt_int))}

  % C Standard, 5.2.4.2.1: INT_MIN <= -32767 (-(2^15 - 1))
  min_int : Axiom
```

```
    min(range_int) <= -32767

  % C Standard, 5.2.4.2.1: INT_MAX >= 32767 (2^15 - 1)
  max_int : Axiom
    max(range_int) >= 32767

  int_binary : Axiom
    FORALL (i: int):
      range_int(i) IFF (min(range_int) <= i And i <= max(range_int))

End Cpp_int


% 3.9.1 pt 2 Signed Long
%-----------------------
Cpp_long : Theory
Begin

  Importing Cpp_int, finite_sets@finite_sets_minmax[int, <=]

  % Alignment
  %-----------
  align_long : posnat

  % proof status :-)
  % range_long_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_long : non_empty_finite_set[int]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_long)]

  dt_long_exists : Axiom
    Exists (dt : (pod_data_type?[(range_long)])) : true

  % proof status :-)
  % dt_long_TCC1 : TCC Obligation

  dt_long : (pod_data_type?[(range_long)])

  % 3.9.1 pt 2: "There are four signed integer types: signed char,
  % short int, int, and long int. In this list, each type
  % provides at least as much storage as those preceding it
  % in the list." -- We take this to mean that both the value
  % range and the size of the object representation must not
  % be smaller.
  dt_long_size : Axiom
    size(uidt(dt_long)) >= size(uidt(dt_int))
```

```
  long_value_supset_int_value : Axiom
    Forall (i : (range_int)) : range_long(i)

  max_value_bits_long : nat

  % proof status :-)
  % value_bitmask_long_TCC1 : TCC Obligation

  value_bitmask_long : {l : list[Byte] | length(l) = size(uidt(dt_long))}

  % C Standard, 5.2.4.2.1: LONG_MIN <= -2147483647 (-(2^31 - 1))
  min_long : Axiom
    min(range_long) <= -2147483647

  % C Standard, 5.2.4.2.1: LONG_MAX >= 2147483647 (2^31 - 1)
  max_long : Axiom
    max(range_long) >= 2147483647

  long_binary : Axiom
    FORALL (i: int):
      range_long(i) IFF (min(range_long) <= i And i <= max(range_long))

End Cpp_long


% Signed Long Long
% (not in the mentioned version of the C++ standard)
%-----------------------------------------------------------------
Cpp_longlong : Theory
Begin

  Importing Cpp_long, finite_sets@finite_sets_minmax[int, <=]

  % Alignment
  %------------
  align_longlong : posnat

  % proof status :-)
  % range_longlong_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_longlong : non_empty_finite_set[int]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_longlong)]

  dt_longlong_exists : Axiom
    Exists (dt : (pod_data_type?[(range_longlong)])) : true

  % proof status :-)
```

```
  % dt_longlong_TCC1 : TCC Obligation

  dt_longlong : (pod_data_type?[(range_longlong)])

  % Following 3.9.1 pt 2 in principle we define:
  dt_longlong_size : Axiom
    size(uidt(dt_longlong)) >= size(uidt(dt_long))

  longlong_value_supset_long_value : Axiom
    Forall (i : (range_long)) : range_longlong(i)

  max_value_bits_longlong : nat

  % proof status :-)
  % value_bitmask_longlong_TCC1 : TCC Obligation

  value_bitmask_longlong : {l : list[Byte] | length(l) =
    size(uidt(dt_longlong))}

  % C Standard, 5.2.4.2.1: LLONG_MIN <= -92233720368547807 (-(2^63 - 1))
  min_longlong : Axiom
    min(range_longlong) <= -92233720368547807

  % C Standard, 5.2.4.2.1: LLONG_MAX >= 92233720368547807 (2^63 - 1)
  max_longlong : Axiom
    max(range_longlong) >= 92233720368547807

  longlong_binary : Axiom
    FORALL (i: int):
      range_longlong(i) IFF
        (min(range_longlong) <= i And i <= max(range_longlong))

End Cpp_longlong


% 3.9.1 pt 3 Unsigned Short Integer
%=================================
Cpp_ushort : Theory
Begin

  Importing Cpp_short, finite_sets@finite_sets_minmax[nat, <=]

  % Alignment
  %-----------
  % 3.9.1 pt 3 unsigned integers shall have the same alignment as the
  % corresponding signed integer
  align_ushort : posnat = align_short

  % proof status :-)
  % range_ushort_TCC1 : TCC Obligation

  % Range of values
```

```
  %-----------------
  range_ushort : non_empty_finite_set[nat]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_ushort)]

  dt_ushort_exists : Axiom
    Exists (dt : (pod_data_type?[(range_ushort)])) : true

  % proof status :-)
  % dt_ushort_TCC1 : TCC Obligation

  dt_ushort : (pod_data_type?[(range_ushort)])

  % 3.9.1 pt 3 unsigned integers shall have the same size as the
  % corresponding signed integer
  dt_ushort_size : Axiom
    size(uidt(dt_ushort)) = size(uidt(dt_short))

  % 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a
  % subrange of the corresponding unsigned integer type, and the
  % value representation of each corresponding signed / unsigned
  % type shall be the same."
  ushort_value_supset_short_value : Axiom
    Forall (i : (range_short)) : i >= 0 Implies range_ushort(i)

  % proof status :-)
  % ushort_represenation_same_for_positive_short_TCC1 : TCC Obligation

  ushort_represenation_same_for_positive_short : Axiom
    Forall (i : (range_short), a : Address) : i >= 0 Implies
      to_byte(dt_ushort)(i, a) = to_byte(dt_short)(i, a)

  % Bits in the value representation
  %--------------------------------
  max_value_bits_ushort : posnat

  % proof status :-)
  % value_bitmask_ushort_TCC1 : TCC Obligation

  value_bitmask_ushort : {l : list[Byte] | length(l) = size(uidt(dt_ushort))}

  % C Standard, 5.2.4.2.1: USHRT_MAX >= 65535 (2^16 - 1)
  max_ushort_bits : Axiom
    max_value_bits_ushort >= 16

  % 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
  % modulo 2^n where n is the number of bits in the value
  % representation ..."
  binary_range_ushort : Axiom
    Forall (n : nat) :
```

```
    range_ushort(n) IFF n <= max(range_ushort)

END Cpp_ushort


% 3.9.1 pt 3 Unsigned Integer
%=================================
Cpp_uint : THEORY
BEGIN

  IMPORTING Cpp_int, finite_sets@finite_sets_minmax[nat, <=]

  % Alignment
  %-----------
  % 3.9.1 pt 3 unsigned integers shall have the same alignment as the
  % corresponding signed integer
  align_uint : posnat = align_int

  % proof status :-)
  % range_uint_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_uint : non_empty_finite_set[nat]

  % (Un)interpreted Datatype
  %-------------------------
  IMPORTING Interpreted_Data[(range_uint)]

  dt_uint_exists : AXIOM
    EXISTS (dt : (pod_data_type?[(range_uint)])) : TRUE

  % proof status :-)
  % dt_uint_TCC1 : TCC Obligation

  dt_uint : (pod_data_type?[(range_uint)])

  % 3.9.1 pt 3 unsigned integers shall have the same size as the
  % corresponding signed integer
  dt_uint_size : AXIOM
    size(uidt(dt_uint)) = size(uidt(dt_int))

  % 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a
  % subrange of the corresponding unsigned integer type, and the
  % value representation of each corresponding signed / unsigned
  % type shall be the same."
  uint_value_supset_int_value : AXIOM
    FORALL (i : (range_int)) : i >= 0 IMPLIES range_uint(i)

  % proof status :-)
  % uint_represenation_same_for_positive_int_TCC1 : TCC Obligation
```

```
  uint_represenation_same_for_positive_int : Axiom
    Forall (i : (range_int), a : Address) : i >= 0 Implies
      to_byte(dt_uint)(i, a) = to_byte(dt_int)(i, a)

  % Bits in the value representation
  %-----------------------------------
  max_value_bits_uint : posnat

  % proof status :-)
  % value_bitmask_uint_TCC1 : TCC Obligation

  value_bitmask_uint : {l : list[Byte] | length(l) = size(uidt(dt_uint))}

  % C Standard, 5.2.4.2.1: UINT_MAX >= 65535 (2^16 - 1)
  max_uint_bits : Axiom
    max_value_bits_uint >= 16

  max_uint : Axiom
    max(range_uint) >= (2^max_value_bits_uint) - 1

  % 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
  % modulo 2^n where n is the number of bits in the value
  % representation ..."
  binary_range_uint : Axiom
    Forall (n : nat) :
      range_uint(n) IFF n <= max(range_uint)

End Cpp_uint


% 3.9.1 pt 3 Unsigned Long
%=========================
Cpp_ulong : Theory
Begin

  Importing Cpp_long, finite_sets@finite_sets_minmax[nat, <=]

  % Alignment
  %-----------
  % 3.9.1 pt 3 unsigned integers shall have the same alignment as the
  % corresponding signed integer
  align_ulong : posnat = align_long

  % proof status :-)
  % range_ulong_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_ulong : non_empty_finite_set[nat]

  % (Un)interpreted Datatype
  %-------------------------
```

```
Importing Interpreted_Data[(range_ulong)]

dt_ulong_exists : Axiom
  Exists (dt : (pod_data_type?[(range_ulong)])) : true

% proof status :-)
% dt_ulong_TCC1 : TCC Obligation

dt_ulong : (pod_data_type?[(range_ulong)])

% 3.9.1 pt 3 unsigned integers shall have the same size as the
% corresponding signed integer
dt_ulong_size : Axiom
  size(uidt(dt_ulong)) = size(uidt(dt_long))

% 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a
% subrange of the corresponding unsigned integer type, and the
% value representation of each corresponding signed / unsigned
% type shall be the same."
ulong_value_supset_long_value : Axiom
  Forall (i : (range_long)) : i >= 0 Implies range_ulong(i)

% proof status :-)
% ulong_represenation_same_for_positive_long_TCC1 : TCC Obligation

ulong_represenation_same_for_positive_long : Axiom
  Forall (i : (range_long), a : Address) : i >= 0 Implies
    to_byte(dt_ulong)(i, a) = to_byte(dt_long)(i, a)

% Bits in the value representation
%-----------------------------------
max_value_bits_ulong : posnat

% proof status :-)
% value_bitmask_ulong_TCC1 : TCC Obligation

value_bitmask_ulong : {l : list[Byte] | length(l) = size(uidt(dt_ulong))}

% C Standard, 5.2.4.2.1: ULONG_MAX >= 4294967295 (2^32 - 1)
max_ulong_bits : Axiom
  max_value_bits_ulong >= 32

max_ulong : Axiom
  max(range_ulong) >= (2^max_value_bits_ulong) - 1

% 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
% modulo 2^n where n is the number of bits in the value
% representation ..."
binary_range_ulong : Axiom
  Forall (n : nat) :
    range_ulong(n) IFF n <= max(range_ulong)
```

```
End Cpp_ulong

% 3.9.1 pt 3 Unsigned Long Long
%==============================
Cpp_ulonglong : Theory
Begin

  Importing Cpp_longlong, finite_sets@finite_sets_minmax[nat, <=]

  % Alignment
  %-----------
  % 3.9.1 pt 3 unsigned integers shall have the same alignment as the
  % corresponding signed integer
  align_ulonglong : posnat = align_longlong

  % proof status :-)
  % range_ulonglong_TCC1 : TCC Obligation

  % Range of values
  %-----------------
  range_ulonglong : non_empty_finite_set[nat]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_ulonglong)]

  dt_ulonglong_exists : Axiom
    Exists (dt : (pod_data_type?[(range_ulonglong)])) : true

  % proof status :-)
  % dt_ulonglong_TCC1 : TCC Obligation

  dt_ulonglong : (pod_data_type?[(range_ulonglong)])

  % 3.9.1 pt 3 unsigned integers shall have the same size as the
  % corresponding signed integer
  dt_ulonglong_size : Axiom
    size(uidt(dt_ulonglong)) = size(uidt(dt_longlong))

  % 3.9.1 pt 3 "The range of nonnegative values of a signed integer type is a
  % subrange of the corresponding unsigned integer type, and the
  % value representation of each corresponding signed / unsigned
  % type shall be the same."
  ulonglong_value_supset_longlong_value : Axiom
    Forall (i : (range_longlong)) : i >= 0 Implies range_ulonglong(i)

  % proof status :-)
  % ulonglong_represenation_same_for_positive_longlong_TCC1 : TCC Obligation

  ulonglong_represenation_same_for_positive_longlong : Axiom
    Forall (i : (range_longlong), a : Address) : i >= 0 Implies
      to_byte(dt_ulonglong)(i, a) = to_byte(dt_longlong)(i, a)
```

```
  % Bits in the value representation
  %-----------------------------------
  max_value_bits_ulonglong : posnat

  % proof status :-)
  % value_bitmask_ulonglong_TCC1 : TCC Obligation

  value_bitmask_ulonglong : {l : list[Byte] | length(l) =
    size(uidt(dt_ulonglong))}

  % C Standard, 5.2.4.2.1: ULLONG_MAX >= 18446744073709551615 (2^64 - 1)
  max_ulonglong_bits : Axiom
    max_value_bits_ulonglong >= 64

  max_ulonglong : Axiom
    max(range_ulonglong) >= (2^max_value_bits_ulonglong) - 1

  % 3.9.1 pt 4 "Unsigned integers [...] shall obey the laws of arithmetic
  % modulo 2^n where n is the number of bits in the value
  % representation ..."
  binary_range_ulonglong : Axiom
    Forall (n : nat) :
      range_ulonglong(n) IFF n <= max(range_ulonglong)

End Cpp_ulonglong


% 3.9 pt 8 Float
%================
Cpp_float : Theory
Begin

  Importing Extended_Real

  % Alignment
  %-----------
  align_float : posnat

  % Value Range
  %-------------
  % subset of number so that we can eventually formalise not-a-thing
  % and +-infinity values

  % proof status :-)
  % range_float_TCC1 : TCC Obligation

  range_float : non_empty_finite_set[extended_real]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_float)]
```

```
  dt_float_exists : Axiom
    Exists (dt_float : (pod_data_type?[(range_float)])) : True

  % proof status :-)
  % dt_float_TCC1 : TCC Obligation

  dt_float : (pod_data_type?[(range_float)])

End Cpp_float

% 3.9 pt 8 Double
%==================
Cpp_double : Theory
Begin

  Importing Cpp_float, Extended_Real

  % Alignment
  %------------
  align_double : posnat

  % Value Range
  %--------------
  % subset of number so that we can eventually formalise not-a-thing
  % and +-infinity values

  range_double : non_empty_finite_set[extended_real]

  % (Un)interpreted Datatype
  %------------------------------
  Importing Interpreted_Data[(range_double)]

  dt_double_exists : Axiom
    Exists (dt_double : (pod_data_type?[(range_double)])) : True

  % proof status :-)
  % dt_double_TCC1 : TCC Obligation

  dt_double : (pod_data_type?[(range_double)])

  % 3.9.1 pt 8 precision float <= double <= long double
  % float values is subset of double values is subset of
  % long double values
  double_precision : Axiom
    Forall (f : (range_float)) : real?(f) Implies range_double(f)


End Cpp_double


% 3.9 pt 8 Long Double
```

```
%=======================
Cpp_longdouble : Theory
Begin

  Importing Cpp_double, Extended_Real

  % Alignment
  %------------
  align_longdouble : posnat

  % Value Range
  %--------------
  % subset of number so that we can eventually formalise not-a-thing
  % and +-infinity values

  range_longdouble : non_empty_finite_set[extended_real]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_longdouble)]

  dt_longdouble_exists : Axiom
    Exists (dt_longdouble : (pod_data_type?[(range_longdouble)])) : True

  % proof status :-)
  % dt_longdouble_TCC1 : TCC Obligation

  dt_longdouble : (pod_data_type?[(range_longdouble)])

  % 3.9.1 pt 8 precision float <= double <= long double
  % float values is subset of double values is subset of
  % long double values
  longdouble_precision : Axiom
    Forall (f : (range_double)) : real?(f) Implies range_longdouble(f)

End Cpp_longdouble


% 3.9.1 pt 6 Bool
%=================
Cpp_bool : Theory
Begin

  Importing Cpp_Deep_Types

  % Alignment
  %------------
  align_bool : posnat

  % !! To do:
  % I (Marcus) read the standard as follows; you have bool objects which
  % are at least byte aligned. Then there are bitfields where single bits
```

```
  % can represent a bool. The above is only for the objects of type bool.
  % Compilers that pack multiple bools in one byte make use of bitfields.

  % proof status :-)
  % range_bool_TCC1 : TCC Obligation

  % Value Range
  %-------------
  range_bool : non_empty_finite_set[bool] = fullset[bool]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[bool]

  dt_bool_exists : Axiom
    Exists (dt_bool : (pod_data_type?[bool])) : True

  % proof status :-)
  % dt_bool_TCC1 : TCC Obligation

  dt_bool : (pod_data_type?[bool])

  % 3.9.1 pt 6 bool
  % bool values behaves as integral type (true < false)
  min_value(typ : (bool?)) : bool = true
  max_value(typ : (bool?)) : bool = false

End Cpp_bool


% 3.9.1 pt 5 Wchar_t
%====================
Cpp_wchar_t : Theory
Begin

  Importing Cpp_char, Cpp_longlong

  % Alignment
  %-----------
  align_wchar_t : posnat

  % 3.9.1 pt 5 "Type wchar_t shall have the same size, signedness and alignment
  % requirements as one other the other integral types, called its
  % underlying type."
  % signed types have the same alignment as their corresponding unsigned types
  wchar_t_alignment : Axiom
    align_wchar_t = align_char Or
    align_wchar_t = align_short Or
    align_wchar_t = align_int Or
    align_wchar_t = align_long Or
    align_wchar_t = align_longlong
```

```
  % proof status :-)
  % range_wchar_t_TCC1 : TCC Obligation

  % Value Range
  %-------------
  range_wchar_t : non_empty_finite_set[int]

  % (Un)interpreted Datatype
  %-------------------------
  Importing Interpreted_Data[(range_wchar_t)]

  dt_wchar_t_exists : Axiom
    Exists (dt_wchar_t : (pod_data_type?[(range_wchar_t)])) : True

  % proof status :-)
  % dt_wchar_t_TCC1 : TCC Obligation

  dt_wchar_t : (pod_data_type?[(range_wchar_t)])

  % proof status :-)
  % value_bitmask_wchar_t_TCC1 : TCC Obligation

  value_bitmask_wchar_t : {l : list[Byte] | length(l) = size(uidt(dt_wchar_t))}

  % 3.9.1 pt 5 "Type wchar_t shall have the same size, signedness and alignment
  % requirements as one other the other integral types, called its
  % underlying type."
  % signed types have the same size as their corresponding unsigned types
  dt_wchar_t_size : Axiom
    size(uidt(dt_wchar_t)) = size(uidt(dt_char)) Or
    size(uidt(dt_wchar_t)) = size(uidt(dt_short)) Or
    size(uidt(dt_wchar_t)) = size(uidt(dt_int)) Or
    size(uidt(dt_wchar_t)) = size(uidt(dt_long)) Or
    size(uidt(dt_wchar_t)) = size(uidt(dt_longlong))

End Cpp_wchar_t


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% A Deep Embedding of non union; non class C++ Types
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The primary uses of the following deep embedding of C++ datatypes are
% - thepolymorphic specificaiton of expressions on fundamental types,
% - the polymorphic specification of pointers and of references
%
% The below deep embedding does not formalise class and union members as these
% give rise to unlimited recursion. The following example illustrates this
% recursion:
%
```

```
% struct A {
% A* pt_A;
% };


Cpp_Deep_Types : THEORY
BEGIN

  Importing More_List_Props

  Access_Specifier : Type = {public, private, protected}

  Cpp_Type_ : DATATYPE
  BEGIN
    % fundamental types (3.9.1)
    %----------------------------
    uchar : uchar?
    schar : schar?
    char : char?

    short : short?
    int : int?
    long : long?
    longlong : longlong?

    ushort : ushort?
    uint : uint?
    ulong : ulong?
    ulonglong : ulonglong?

    wchar_t : wchar_t?

    bool : bool?

    float : float?
    double : double?
    longdouble : longdouble?

    void : void?

    % compound types (3.9.2)
    %------------------------
    % array types (8.3.4)
    % 8.3.4 pt 1 [...] If the constant-expression is present, it shall be an
    % integral constant expression and its value shall be greater
    % than zero. [...])
    % Thus, it is possible to encode arrays of unknown size as arrays of size 0
    array_type(typ : Cpp_Type_, size : nat) : array?

    % function types (8.3.5)
    function_type(ret_typ : Cpp_Type_,
                  args_typ : list[Cpp_Type_],
```

```
                    var_args : bool) : function?
% member_function : bool) : function?

    % pointer types (8.3.1)
    pointer(typ : Cpp_Type_) : pointer?

    % reference types (8.3.2)
    reference(typ : Cpp_Type_) : reference?

    % classes (clause 9)
    class(name : string,
          abstract : bool) : class?

    % unions (9.5)
    union(name : string) : union?
% members : posnat,
% mutable : [below[members] -> bool],
% access_spec : [below[members] -> Access_Specifier]) : union?

    % 3.9.2 pt 1. static members
    % "Static class members are objects or functions, and pointers to them
    % are ordinary pointers to objects or functions"
    % => Static members are allocated at an object-independent address.
    % Thus, there is no need to include static members into the deep
    % embedding of classes and unions


    % Bit fields are addressed relative to an allocation unit. An Address
    % (e.g., a const reference to a bit field) identifies this allocation unit,
    % the bit_offset addresses the first bits within this allocation unit and
    % the bit_size the size of the bitfield. Therefore, size(uidt(dt_bitf))
    % corresponds to the size of the allocation unit. We modify a bitfield by
    % reading its allocation unit, setting the modified bits (as determined by
    % the mask obtained from to_byte(dt_bitf) and by writing the allocation
    % unit. Reading and writing of the allocation unit is by read list; reading
    % may be omitted if the mask modifies all bits in the allocation unit.
    % Allocation units may overlap. We translate unnamed bit fields and padding
    % into "holes" in the respective allocation units.

    % bit field (9.6)
    bitfield (typ : Cpp_Type_, % integral or enum
              bit_offset : nat, bit_size : posnat) : bitfield?

    % enumerations (7.2)
    enum(name : string,
         typ : Cpp_Type_, % integral
         members : posnat,
         constant_value : [below[members] -> int]) : enum?

    % proof status :-)
    % Cpp_Type__pointer_to_member_eta_TCC1 : TCC Obligation
```

```
    % proof status :-)
    % Cpp_Type__pointer_to_member_eta_TCC2 : TCC Obligation

    % proof status :-)
    % Cpp_Type__typ_pointer_to_member_TCC1 : TCC Obligation

    % pointer-to-member (8.3.3)
    pointer_to_member(typ : (class?),
                      member_typ : Cpp_Type_) : pointer_to_member?

    % CV qualifiers (3.9.3)
    %-----------------------
    % 3.9.3 pt 1 const and volatile qualified types define distinct C++ types
    % ...
    const (typ : Cpp_Type_) : const? % not class, union, array or
                                     % function
    volatile (typ : Cpp_Type_) : volatile? % -> semantic compiler sets cv
                                     % for member accesses from
                                     % within a cv function

  END Cpp_Type_

  % Properties of the subterm relation
  %-----------------------------------
  % proof status :-)
  subterm_less : Lemma
    Forall (t1, t2 : Cpp_Type_) :
      subterm(t1, t2) And t1 /= t2 Implies t1 << t2

  % proof status :-)
  subterm_transitive : Lemma
    Forall (t1, t2, t3 : Cpp_Type_) :
      subterm(t1, t2) And subterm(t2, t3) Implies subterm(t1, t3)

  % Classification of Types
  %=========================

  % proof status :-)
  % c_TCC1 : TCC Obligation

  % a possibly cv qualified type
  c(p : PRED[Cpp_Type_])(typ : Cpp_Type_) : bool =
    p(typ) Or (const?(typ) And p(typ(typ)))

  % proof status :-)
  % v_TCC1 : TCC Obligation

  v(p : PRED[Cpp_Type_])(typ : Cpp_Type_) : bool =
    p(typ) Or (volatile?(typ) And p(typ(typ)))

  % proof status :-)
  % cv_TCC1 : TCC Obligation
```

```
  % proof status :-)
  % cv_TCC2 : TCC Obligation

  cv(p : PRED[Cpp_Type_])(typ : Cpp_Type_) : bool =
    c(p)(typ) Or v(p)(typ) Or
      (const?(typ) And volatile?(typ(typ)) And p(typ(typ(typ))))

  % proof status :-)
  cv_implied : Lemma
    Forall (p1, p2 : PRED[Cpp_Type_], typ : Cpp_Type_) :
      subset?(p1, p2) And cv(p1)(typ) Implies cv(p2)(typ)

  % incomplete types (3.9 pt 6)
  incomplete_type?(typ : Cpp_Type_) : bool =
% a class may become complete while the type remains the same
% (class?(typ) And size(typ) = 0) Or
    (array?(typ) And size(typ) = 0) Or
     void?(typ)

  % object_types (3.9 pt 9)
  object_type?(typ : Cpp_Type_) : bool =
    Not cv(function?)(typ) And Not cv(reference?)(typ) And Not cv(void?)(typ)

  % Fundamental Types
  %-------------------

  % signed integer types (3.9.1 pt 2)
  signed_integer?(typ : Cpp_Type_) : bool =
    schar?(typ) Or short?(typ) Or int?(typ) Or long?(typ) Or longlong?(typ)

  % unsigned integer types (3.9.1 pt 3)
  unsigned_integer?(typ : Cpp_Type_) : bool =
    uchar?(typ) Or ushort?(typ) or uint?(typ) or ulong?(typ) or ulonglong?(typ)

  % integral types (3.9.1 pt 7)
  % we handle bool separately because the standard defines several
  % special cases for bool expressions.
  non_bool_integral?(typ : Cpp_Type_) : bool =
    char?(typ) Or wchar_t?(typ) Or
      signed_integer?(typ) Or unsigned_integer?(typ)

  integral?(typ : Cpp_Type_) : bool =
    bool?(typ) Or non_bool_integral?(typ)

  % floating point types (3.9.1 pt 8)
  floating_point?(typ : Cpp_Type_) : bool =
    float?(typ) Or double?(typ) Or longdouble?(typ)

  % arithmetic types (3.9.1 pt 8)
  arithmetic?(typ : Cpp_Type_) : bool =
    integral?(typ) Or floating_point?(typ)
```

```
  fundamental?(typ : Cpp_Type_) : bool =
    arithmetic?(typ) Or void?(typ)

  % scalar types (3.9 p10)
  scalar?(typ : Cpp_Type_) : bool =
    arithmetic?(typ) Or enum?(typ) Or
      pointer?(typ) Or pointer_to_member?(typ)

  % pod stuct / union types (3.9 pt 10)
  pod_struct?(typ : (class?)) : bool
  pod_union?(typ : (class?)) : bool

  % proof status :-)
  % pod?_TCC1 : TCC Obligation

  % pod types (3.9 pt 10)
  pod?(t : Cpp_Type_) : bool =
    scalar?(t) Or
    (class?(t) And (pod_struct?(t) Or pod_union?(t))) Or
    (array?(t) And (scalar?(typ(t)) Or
       (class?(typ(t)) Implies (pod_struct?(typ(t)) Or pod_union?(typ(t))))))

  % Further Classifications
  %------------------------

  non_bool_integral_enum?(t : Cpp_Type_) : bool =
    non_bool_integral?(t) Or enum?(t)

  % cpp types for which we define an interpreted data type
  % - void, functions, arrays, classes and unions remain uninterpreted
  interpreted?(t : Cpp_Type_) : bool =
    scalar?(t) Or reference?(t) Or bitfield?(t)

  % Range of Values
  %================

  % We use the PVS type number (more precisely predicates of this type)
  % to encode the value range of fundamental types. Reals do not suffice
  % because bool is an integral type and we do not want to define
  % a real constant for true and false and because floating point types
  % may encode positive and negative infinity and not-a-thing values

  % 3.9.1 pt 9 void
  range_void : finite_set[int] = emptyset[int]

  Importing Uninterpreted_Data

  % proof status :-)
  % uidt_void_TCC1 : TCC Obligation

  uidt_void : (uninterpreted_data_type?) =
```

```
    (# size := 0, valid? := Lambda (l : list[Byte], a : Address) : False #)

End Cpp_Deep_Types


% To do: don't we need a theory instantiation of the above theories rather
% than their importing?
Cpp_Integral_Types : Theory
Begin

  Importing Interpreted_Data,
            Cpp_ushort, Cpp_uint, Cpp_ulong, Cpp_ulonglong,
            Cpp_longdouble, Cpp_bool, Cpp_wchar_t,
            finite_sets@finite_sets_minmax[real, <=]

  % proof status :-)
  % range_integral_TCC1 : TCC Obligation

  % proof status :-)
  % range_integral_TCC2 : TCC Obligation

  % proof status :-)
  % range_integral_TCC3 : TCC Obligation

  % proof status :-)
  % range_integral_TCC4 : TCC Obligation

  % proof status :-)
  % range_integral_TCC5 : TCC Obligation

  % proof status :-)
  % range_integral_TCC6 : TCC Obligation

  range_integral(typ : (non_bool_integral?)) : non_empty_finite_set[int] =
    Cases typ Of
      uchar : range_uchar,
      schar : range_schar,
      char : range_char,

      short : range_short,
      int : range_int,
      long : range_long,
      longlong : range_longlong,

      ushort : range_ushort,
      uint : range_uint,
      ulong : range_ulong,
      ulonglong : range_ulonglong,

      wchar_t : range_wchar_t
    EndCases
```

```
% proof status :-)
% range_floating_point_TCC1 : TCC Obligation

range_floating_point(typ : (floating_point?)) :
    non_empty_finite_set[extended_real] =
  Cases typ Of
    float : range_float,
    double : range_double,
    longdouble : range_longdouble
  EndCases

% proof status :-)
% dt_integral_TCC1 : TCC Obligation

% proof status :-)
% dt_integral_TCC2 : TCC Obligation

% proof status :-)
% dt_integral_TCC3 : TCC Obligation

% proof status :-)
% dt_integral_TCC4 : TCC Obligation

% proof status :-)
% dt_integral_TCC5 : TCC Obligation

% proof status :-)
% dt_integral_TCC6 : TCC Obligation

% proof status :-)
% dt_integral_TCC7 : TCC Obligation

% proof status :-)
% dt_integral_TCC8 : TCC Obligation

% proof status :-)
% dt_integral_TCC9 : TCC Obligation

% proof status :-)
% dt_integral_TCC10 : TCC Obligation

% proof status :-)
% dt_integral_TCC11 : TCC Obligation

% proof status :-)
% dt_integral_TCC12 : TCC Obligation

dt_integral(typ : (non_bool_integral?)) :
    (pod_data_type?[(range_integral(typ))]) =
  Cases typ Of
    uchar : dt_uchar,
```

```
    schar : dt_schar,
    char : dt_char,

    short : dt_short,
    int : dt_int,
    long : dt_long,
    longlong : dt_longlong,

    ushort : dt_ushort,
    uint : dt_uint,
    ulong : dt_ulong,
    ulonglong : dt_ulonglong,

    wchar_t : dt_wchar_t
  EndCases

% proof status :-)
% dt_floating_point_TCC1 : TCC Obligation

% proof status :-)
% dt_floating_point_TCC2 : TCC Obligation

% proof status :-)
% dt_floating_point_TCC3 : TCC Obligation

dt_floating_point(typ : (floating_point?)) :
    (pod_data_type?[(range_floating_point(typ))]) =
  Cases typ Of
    float : dt_float,
    double : dt_double,
    longdouble : dt_longdouble
  EndCases

% proof status :-)
% range_integral_unsigned_TCC1 : TCC Obligation

% proof status :-)
range_integral_unsigned : Judgement
  range_integral(typ : (unsigned_integer?)) Has_Type
    non_empty_finite_set[nat]


% proof status :-)
% max_value_TCC1 : TCC Obligation

% proof status :-)
% max_value_TCC2 : TCC Obligation

% 3.9.1 pt 8 "... numeric_limits shall specify the maximum and minimum values
% for each arithmetic type ..."
max_value(typ : (non_bool_integral?)) : int =
  max(range_integral(typ))
```

```
  % proof status :-)
  % min_value_TCC1 : TCC Obligation

  min_value(typ : (non_bool_integral?)) : int =
    min(range_integral(typ))

  % proof status :-)
  max_value_unsigned : Judgement
    max_value(typ : (unsigned_integer?)) Has_Type nat

  % proof status :-)
  min_value_unsigned : Judgement
    min_value(typ : (unsigned_integer?)) Has_Type nat

  % proof status :-)
  % max_value_bits_TCC1 : TCC Obligation

  % proof status :-)
  % max_value_bits_TCC2 : TCC Obligation

  % proof status :-)
  % max_value_bits_TCC3 : TCC Obligation

  % proof status :-)
  % max_value_bits_TCC4 : TCC Obligation

  % proof status :-)
  % max_value_bits_TCC5 : TCC Obligation

  % proof status :-)
  % max_value_bits_TCC6 : TCC Obligation

  max_value_bits(typ : (cv(unsigned_integer?))) : Recursive posnat =
    Cases typ Of
      uchar : max_value_bits_uchar,
      ushort : max_value_bits_ushort,
      uint : max_value_bits_uint,
      ulong : max_value_bits_ulong,
      ulonglong : max_value_bits_ulonglong,
      const(t) : max_value_bits(t),
      volatile(t) : max_value_bits(t)
    EndCases
   Measure typ by <<

End Cpp_Integral_Types


Cpp_Deep_Type_Wellformedness : Theory
Begin

  Importing Cpp_Integral_Types
```

```
  % Constraints on Recursive Definition of Types
  %=============================================

  % proof status :-)
  % no_pointers_to_bitfield?_TCC1 : TCC Obligation

  % Pointers
  %----------
  % 8.3.1 pt 4 no pointers to bitfields
  no_pointers_to_bitfield?(t : Cpp_Type_) : bool =
    Not (pointer?(t) And cv(bitfield?)(typ(t)))

  % 8.3.2 pt 4 no pointers to ref
  no_pointers_to_references?(t : Cpp_Type_) : bool =
    Not (pointer?(t) And reference?(typ(t)))

  % References
  %------------

  % proof status :-)
  % no_cv_references?_TCC1 : TCC Obligation

  % 8.3.2 pt 1 no cv(ref)
  no_cv_references?(t : Cpp_Type_) : bool =
    Not ((const?(t) Or volatile?(t)) And reference?(typ(t)))

  % proof status :-)
  % no_reference_to_reference?_TCC1 : TCC Obligation

  % 8.3.2 pt 4 no ref to ref
  no_reference_to_reference?(t : Cpp_Type_) : bool =
    Not (reference?(t) And reference?(typ(t)))

  % 9.6 pt 1 "Bit fields are packed into some addressable allocation unit"
  % no ref to bitfield
  no_reference_to_bitfields?(t : Cpp_Type_) : bool =
    Not (reference?(t) And cv(bitfield?)(typ(t)))

  % 8.3.2 pt 4 "A reference shall be initialised to point to a valid object"
  % 3.9.1 pt 9 "An object-type is a [...] type that is not [...] a void type."
  % We take this to mean no references to void.
%>> 3.9.1 pt 9 declares functions are no objects, however,
% gcc allows int (&bar)(...)
  no_reference_to_void?(t : Cpp_Type_) : bool =
    Not (reference?(t) And cv(void?)(typ(t)))

  % Pointer To Member
  %------------------

  % 8.3.3 pt 3 no pointer_to_member to static members (trivial)
```

```
  % 8.3.3 pt 3 no pointer_to_member to reference
  no_pointer_to_member_to_reference?(t : Cpp_Type_) : bool =
    Not (pointer_to_member?(t) And reference?(member_typ(t)))

  % 8.3.3 pt 3 no pointer_to_member to cv void
  no_pointer_to_member_to_cv_void?(t : Cpp_Type_) : bool =
    Not (pointer_to_member?(t) And cv(void?)(member_typ(t)))

  % proof status :-)
  % no_array_of_references?_TCC1 : TCC Obligation

  % Arrays
  %---------
  % 8.3.2 pt 4 and 8.3.4 pt 1 no array of ref
  no_array_of_references?(t : Cpp_Type_) : bool =
    array?(t) Implies Not reference?(typ(t))

  % 8.3.4 pt 1 no array to cv void
  no_array_of_cv_void?(t : Cpp_Type_) : bool =
    array?(t) Implies Not cv(void?)(typ(t))

  % 8.3.4 pt 1 no array to function
  no_array_of_function?(t : Cpp_Type_) : bool =
    array?(t) Implies Not function?(typ(t))

  % 8.3.4 pt 1 no array of abstract class type
  no_array_of_abstract_class?(t : Cpp_Type_) : bool =
    array?(t) Implies Not (class?(typ(t)) And abstract(typ(t)))

  % struct { int [] : 5 }; is ill formed
  no_array_of_bitfields?(t : Cpp_Type_) : bool =
    array?(t) Implies Not cv(bitfield?)(typ(t))

  % Functions
  %-----------

  % 8.3.5 pt 4 cv(function) only for nonstatic member functions
  % Cv member functions access members as if they were cv qualified
  % we handle this situation in the semantics compiler by declar
  % members accessed within a cv function to have cv type.
  % We can therefore forbit cv(function)
  no_cv_function?(t : Cpp_Type_) : bool =
    Not ((const?(t) Or volatile?(t)) And function?(typ(t)))

  % To do: semantics compiler: members accessed within cv function must
  % have cv type

  % 8.3.5 pt 2 no void function parameter
  % f(void) is equivalent to f() with an empty list
  no_cv_void_parameter?(t : Cpp_Type_) : bool =
    function?(t) Implies
      every(Lambda (p : Cpp_Type_) : not cv(void?)(p))(args_typ(t))
```

```
% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC1 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC2 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC3 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC4 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC5 : TCC Obligation

% proof status :-)
% no_pointer_or_ref_to_incomplete_array_parameter?_TCC6 : TCC Obligation

% 8.3.5 pt 6 no function parameters of type:
% pointer to array of unknown bound; reference to array of unknown bound
no_pointer_or_ref_to_incomplete_array_parameter?(t : Cpp_Type_) : bool =
  function?(t) Implies every(Lambda (p : Cpp_Type_) :
    Not ((pointer?(p) Or reference?(p)) And
          (array?(typ(p)) And size(typ(p)) > 0)) And
    Not ((const?(p) Or volatile?(p)) And pointer?(typ(p)) And
          (array?(typ(typ(p))) And size(typ(typ(p))) > 0)) And
    Not (const?(p) And volatile?(typ(p)) And pointer?(typ(typ(p))) And
          (array?(typ(typ(typ(p)))) And size(typ(typ(typ(p)))) > 0))
  )(args_typ(t))

% 8.3.5 pt 6 no return type array or function
no_array_return_type?(t : Cpp_Type_) : bool =
  Not (function?(t) And array?(ret_typ(t)))

no_function_return_type?(t : Cpp_Type_) : bool =
  Not (function?(t) And function?(ret_typ(t)))

% To do:
% 8.3.5 pt 6 no incomplete types in return type or function
% (except for member functions)

% Classes and Unions
%--------------------
% no restrictions on element types

% Bit Fields
%------------

% 9.6 pt 3 bit-field shall have integral or enumeration type

% proof status :-)
```

```
  % bitfield_underlying_integral_or_enum_type?_TCC1 : TCC Obligation

  %>> we deviate from the standard by disallowing bools to be stored
  % directly in a bitfield; use bool_to_int conversions instead
  bitfield_underlying_integral_or_enum_type?(t : Cpp_Type_) : bool =
    bitfield?(t) Implies (non_bool_integral_enum?(typ(t)))

  % Enums
  %-------
  % 7.2 "The underlying type [...] is an integral type that can
  % represent all the [enum] values"

  % proof status :-)
  % enum_underlying_integral?_TCC1 : TCC Obligation

  enum_underlying_integral?(t : Cpp_Type_) : bool =
    enum?(t) Implies non_bool_integral?(typ(t))

  % proof status :-)
  % enum_constants?_TCC1 : TCC Obligation

  % proof status :-)
  % enum_constants?_TCC2 : TCC Obligation

  enum_constants?(e : (enum_underlying_integral?)) : bool =
    enum?(e) Implies
      Forall (m : below[members(e)]) :
        range_integral(typ(e))(constant_value(e)(m))

  % CV Qualifiers
  %---------------

  % proof status :-)
  % const_volatile?_TCC1 : TCC Obligation

  % 3.9.3 pt 1 const(volatile(typ)) = volatile(const(typ))
  % (rule out the latter)
  const_volatile?(t : Cpp_Type_) : bool =
    volatile?(t) Implies Not const?(typ(t))

  % proof status :-)
  % const_stutter?_TCC1 : TCC Obligation

  const_stutter?(t : Cpp_Type_) : bool =
    const?(t) Implies Not const?(typ(t))

  volatile_stutter?(t : Cpp_Type_) : bool =
    volatile?(t) Implies Not volatile?(typ(t))

  % proof status :-)
  % cv_array?_TCC1 : TCC Obligation
```

```
% 3.9.3 pt 2 cv(array(typ)) = array(cv(typ)) rule out the first
cv_array?(t : Cpp_Type_) : bool =
  (volatile?(t) Or const?(t)) Implies Not array?(typ(t))

% 3.9.3 pt 3 cv(class(t1, ..., tn)) =>
% cv(t1) .. cv(tn) provided ti is not a reference
% 3.9.3 pt 3 const(class(t1, ..., tn)) =>
% const(t1) .. const(tn) provided ti is not mutable
% We handle this case in the semantics compiler.
no_cv_class?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And class?(typ(t)))

no_cv_union?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And union?(typ(t)))

no_cv_void?(t : Cpp_Type_) : bool =
  Not ((const?(t) Or volatile?(t)) And void?(typ(t)))

% To do: semantics compiler cv(class(t1,...tn))) => cv(ti)

% Combination of individual constraints in a single predicative subtype
%-----------------------------------------------------------------------

Cpp_Type?(typ : Cpp_Type_) : bool =
  Forall (t : Cpp_Type_) : subterm(t, typ) Implies
    no_pointers_to_bitfield?(t) And no_pointers_to_references?(t) And
    no_cv_references?(t) And no_reference_to_reference?(t) And
    no_reference_to_bitfields?(t) And
    no_pointer_to_member_to_reference?(t) And
    no_pointer_to_member_to_cv_void?(t) And no_cv_void_parameter?(t) And
    no_array_of_references?(t) And no_array_of_cv_void?(t) And
    no_array_of_function?(t) And no_array_of_abstract_class?(t) And
    no_pointer_or_ref_to_incomplete_array_parameter?(t) And
    no_array_return_type?(t) And no_function_return_type?(t) And
    bitfield_underlying_integral_or_enum_type?(t) And cv_array?(t) And
    enum_underlying_integral?(t) And enum_constants?(t) And
    const_volatile?(t) And
    const_stutter?(t) And volatile_stutter?(t) And no_cv_class?(t) And
    no_cv_union?(t) And no_cv_function?(t) And no_reference_to_void?(t) And
    no_cv_void?(t) And
    no_array_of_bitfields?(t)

Cpp_Type : Type = (Cpp_Type?)

Cpp_Subtype(P : [Cpp_Type_ -> bool]) : Type = {typ : Cpp_Type | P(typ)}


% proof status :-)
% cv_base_TCC1 : TCC Obligation

% proof status :-)
% cv_base_TCC2 : TCC Obligation
```

```
% proof status :-)
% cv_base_TCC3 : TCC Obligation

% proof status :-)
% cv_base_TCC4 : TCC Obligation

% proof status :-)
% cv_base_TCC5 : TCC Obligation

% Reduce cv qualifiedness from a type
%-----------------------------------
cv_base(typ : Cpp_Type) : Recursive Cpp_Type =
  Cases typ Of
    const(t) : cv_base(t),
    volatile(t) : cv_base(t)
  Else
    typ
  EndCases
 Measure typ by <<

% proof status :-)
subterm_cv_base : Lemma
  Forall (typ : Cpp_Type) :
    subterm(cv_base(typ), typ)

% proof status :-)
cv_base_result : Lemma
  Forall (P : {Q : PRED[Cpp_Type_] | subset?(Q, interpreted?)},
          typ : Cpp_Subtype(cv(P))) :
    P(cv_base(typ))

% proof status :-)
cv_base_result2 : Lemma
  Forall (typ : Cpp_Type) :
    Not const?(typ) And Not volatile?(typ)
  Implies
    cv_base(typ) = typ

% proof status :-)
cv_base_not_const : Lemma
  Forall (typ : Cpp_Type) :
    Not const?(cv_base(typ))

% proof status :-)
cv_base_not_volatile : Lemma
  Forall (typ : Cpp_Type) :
    Not volatile?(cv_base(typ))

% proof status :-)
cv_base_cv : Lemma
  Forall (P : PRED[Cpp_Type_], typ : Cpp_Type) :
```

```
      subset?(P, interpreted?) Implies
        (P(cv_base(typ)) IFF cv(P)(typ))

  % Layout Compatibility
  %=======================

  % proof status :-)
  % layout_compatible?_TCC1 : TCC Obligation

  % (3.9 pt 11) [reflexive] same types are layout compatible
  % symmetric, transitive follows from nature of layout_compatibility?
  layout_compatible? : equivalence[Cpp_Type]

  % proof status :-)
  % layout_compatible_enums_TCC1 : TCC Obligation

  % proof status :-)
  % layout_compatible_enums_TCC2 : TCC Obligation

  % 7.2 pt 7 "Two enumeration types are layout compatible if the have the
  % same underlying type."
  layout_compatible_enums : Axiom
    Forall (e1, e2 : Cpp_Subtype(enum?)) :
      layout_compatible?(e1, e2) IFF typ(e1) = typ(e2)

  % 3.9.3 pt 1 "The cv-qualified or cv-unqualified version of a type [...]
  % shall have the same representation and alignment requirement"
  % We take this to mean they are layout compatible.
  layout_compatible_cv : Axiom
    Forall (t : Cpp_Type) :
      (Cpp_Type?(const(t)) Implies layout_compatible?(t, const(t))) And
      (Cpp_Type?(volatile(t)) Implies layout_compatible?(t, volatile(t))) And
      (Cpp_Type?(const(volatile(t))) Implies
         layout_compatible?(t, const(volatile(t))))

End Cpp_Deep_Type_Wellformedness

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Compound Types
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% 7.2 Enumerations
%==================
Cpp_enum : Theory
Begin
  Importing Cpp_Deep_Type_Wellformedness

  e : Var Cpp_Subtype(enum?)

  % Alignment
```

```
  %------------
  align_enum(e) : posnat

  % Value Range
  %--------------

  % proof status :-)
  % valid_range?_TCC1 : TCC Obligation

  % proof status :-)
  % valid_range?_TCC2 : TCC Obligation

  % 7.2 pt 5 "The underlying type of an enumeration is an integral type that
  % can represent all the enumerator values"
  valid_range?(e)(r : non_empty_finite_set[int]) : bool =
    subset?(r, range_integral(typ(e))) And
      Forall (member : below[members(e)]) : r(constant_value(e)(member))

  % To do: ?
  % Currently the smallest range of an enum is comprised only of the discrete
  % enum constants. However, 7.2 pt 6 allows all values between emin and emax
  % or more precisely in the surrounding bitfield bmin, bmax.
  % 7.2 pt 6 bitfield
  % bmin <= emin <= emax <= bmax for a bitfield b

  % proof status :-)
  % range_enum_TCC1 : TCC Obligation

  range_enum(e) : (valid_range?(e))

  % (Un-)interpreted Datatype
  %--------------------------

  dt_enum_exists : Axiom
    Exists (dt : (pod_data_type?[(range_enum(e))])) : true

  % Remark: the datatype of the underlying type --- dt(typ(e)) --- should be
  % a valid model for dt(e) and all possible ranges.

  % proof status :-)
  % dt_enum_TCC1 : TCC Obligation

  dt_enum(e) : (pod_data_type?[(range_enum(e))])

  % proof status :-)
  % value_bitmask_enum_TCC1 : TCC Obligation

  value_bitmask_enum(e) : {l : list[Byte] | length(l) = size(uidt(dt_enum(e)))}

  % proof status :-)
  % enum_size_TCC1 : TCC Obligation
```

```
  % proof status :-)
  % enum_size_TCC2 : TCC Obligation

  % 7.2 pt 5 "The value of sizeof() applied to an enumeration type [...] is
  % the value of sizeof() applied to the unterlying type. "
  enum_size : Axiom
    size(uidt(dt_enum(e))) = size(uidt(dt_integral(typ(e))))

End Cpp_enum


Cpp_pointer : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  p : Var Cpp_Subtype(pointer?)
  i : Var int

  % Alignment
  %------------
  align_ptr(p) : posnat

  % Value Range
  %--------------
  Pointer_Base_Type : Type

  % Alternative models:
  %~~~~~~~~~~~~~~~~~~~~
  % 1) Pointer_Base_Type = Memory_Address
  % - no bounds checking; + no hidden bits required ;
  % + size = l2(Max_Address) possible
  %
  % 2) Pointer_Base_Type =
  % [# base : Memory_Address, size : posnat, offset : int #]
  % + bounds checking possible ;
  % - requires hidden bits or size > l2(Max_Address)

  % To do: implement the above two models

  % 3.9.2 pt 3 pointer value is address of a byte in memory or a null pointer
  range_pointer(p) : PRED[lift[Pointer_Base_Type]]

  null?(ptr_val : lift[Pointer_Base_Type]) : bool = bottom?(ptr_val)

  % we assert that the null pointer is in the semantics range of any pointer
  null_every_range : Axiom
    Forall (p) : range_pointer(p)(bottom)

  % (Un-)interpreted Datatype
  %============================
```

```
  dt_pointer_exists : Axiom
    Exists (dt : (pod_data_type?[(range_pointer(p))])) : true

  % proof status :-)
  % dt_pointer_TCC1 : TCC Obligation

  dt_pointer(p) : (pod_data_type?[(range_pointer(p))])

  % proof status :-)
  % pointers_to_layout_compatible_range_TCC1 : TCC Obligation

  % proof status :-)
  % pointers_to_layout_compatible_range_TCC2 : TCC Obligation

  % proof status :-)
  % pointers_to_layout_compatible_range_TCC3 : TCC Obligation

  % proof status :-)
  % pointers_to_layout_compatible_range_TCC4 : TCC Obligation

  % 3.9.2 pt 3 pointers to layout-compatible (possibly cv_qualified) types
  % shall have the same value representation and alignment
  % requirements
  pointers_to_layout_compatible_range : Axiom
    Forall (p1, p2 : Cpp_Subtype(pointer?)) :
      layout_compatible?(typ(p1), typ(p2)) Implies
        range_pointer(p1) = range_pointer(p2)

  % proof status :-)
  % pointers_to_layout_compatible_dt_TCC1 : TCC Obligation

  pointers_to_layout_compatible_dt : Axiom
    Forall (p1, p2 : Cpp_Subtype(pointer?)) :
      layout_compatible?(typ(p1), typ(p2)) Implies
        dt_pointer(p1) = dt_pointer(p2)

  pointers_to_layout_compatible_alignment : Axiom
    Forall (p1, p2 : Cpp_Subtype(pointer?)) :
      layout_compatible?(typ(p1), typ(p2)) Implies
        align_ptr(p1) = align_ptr(p2)

  % 3.9.2 pt 4 "cv? void* and void* shall have the same representation
  % and alignment requirements as a cv? char*"
  void_pointer_char_pointer_range : Axiom
    Forall (p1, p2 : Cpp_Subtype(pointer?)) :
      void?(typ(p1)) And char?(typ(p2))
    Implies
      range_pointer(p1) = range_pointer(p2)

  % proof status :-)
  % void_pointer_char_pointer_dt_TCC1 : TCC Obligation
```

```
void_pointer_char_pointer_dt : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    void?(typ(p1)) And char?(typ(p2))
  Implies
    dt_pointer(p1) = dt_pointer(p2)

void_pointer_char_pointer_alignment : Axiom
  Forall (p1, p2 : Cpp_Subtype(pointer?)) :
    void?(typ(p1)) And char?(typ(p2))
  Implies
    align_ptr(p1) = align_ptr(p2)

% Pointer Arithmetic
%====================

% Address of Pointer and Arithmetic
%--------------------------------

ptyp : Var Cpp_Subtype(cv(pointer?))

not_null?(ptyp)(ptr_val : (range_pointer(cv_base(ptyp)))) : bool = Not null?(ptr_val)

address_of(ptyp)(ptr_val : (range_pointer(cv_base(ptyp)))) : lift[Memory_Address]

address_of_spec : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) : not_null?(ptyp)(ptr_val)
    Implies up?(address_of(ptyp)(ptr_val))

address_of_spec2 : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    null?(ptr_val) Implies bottom?(address_of(ptyp)(ptr_val))

% proof status :-)
% add_TCC1 : TCC Obligation

% Integer addition for pointers
add(ptyp)(ptr_val : (range_pointer(cv_base(ptyp))), i : int) : (range_pointer(cv_base(ptyp)))

add_spec : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    not_null?(ptyp)(ptr_val) Iff not_null?(ptyp)(add(ptyp)(ptr_val, i))

% proof status :-)
% add_spec2_TCC1 : TCC Obligation

% proof status :-)
% add_spec2_TCC2 : TCC Obligation

add_spec2 : Axiom
  Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
    not_null?(ptyp)(ptr_val)
  Implies
```

```
      down(address_of(ptyp)(add(ptyp)(ptr_val, i))) =
        down(address_of(ptyp)(ptr_val)) + i

  % proof status :-)
  add_simplification : Lemma
    Forall (ptr_val : (range_pointer(cv_base(ptyp))), i, j : int) :
      address_of(ptyp)(add(ptyp)(add(ptyp)(ptr_val, i), j)) =
        address_of(ptyp)(add(ptyp)(ptr_val, i + j))

  % Pointer substraction
  % 5.7 pt 6 defines substraction only for pointers of the same array object.
  % 5.9 pt 2 pointers to the same array are comparable

  same_array(ptyp)(ptr_val1, ptr_val2 : (range_pointer(cv_base(ptyp)))) : bool

  same_array_spec : Axiom
    Forall (ptr_val1, ptr_val2 : (range_pointer(cv_base(ptyp)))) :
      not_null?(ptyp)(ptr_val1) And not_null?(ptyp)(ptr_val2)
    Implies
      (same_array(ptyp)(ptr_val1, ptr_val2) IFF ptr_val1 = ptr_val2)

  same_array_spec2 : Axiom
    Forall (ptr_val1, ptr_val2 : (range_pointer(cv_base(ptyp)))) :
      null?(ptr_val1) Or null?(ptr_val2)
    Implies
      same_array(ptyp)(ptr_val1, ptr_val2) = false

  % Array Bounds Checking (Requires Model 2)
  %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  check_bounds(ptyp)(ptr_val : (range_pointer(cv_base(ptyp)))) : bool

  check_bounds_spec : Axiom
    Forall (ptr_val : (range_pointer(cv_base(ptyp)))) :
      null?(ptr_val)
    Implies
      check_bounds(ptyp)(ptr_val) = false

  % Respective model (2) implementations of these functions are:
  %~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  %
  % - add(p)(ptr_val, i) = 'offset += i
  % - address_of(p)(ptr_val) = base + offset
  % - check_bounds(p)(ptr_val) = 0 <= offset < size

  % proof status :-)
  % void_pointer_other_pointer_value_TCC1 : TCC Obligation

  % proof status :-)
  % void_pointer_other_pointer_value_TCC2 : TCC Obligation

  % 3.9.2 pt 4 "A void* shall be able to hold hold any object pointer."
  void_pointer_other_pointer_value : Axiom
```

```
    Forall (p1, p2 : Cpp_Subtype(cv(pointer?)), ptr_val1 : (range_pointer(cv_base(p1)))) :
      void?(typ(cv_base(p1))) And not_null?(p1)(ptr_val1)
    Implies
      Exists (ptr_val2 : (range_pointer(cv_base(p2)))) :
        not_null?(p2)(ptr_val2) And
        down(address_of(p1)(ptr_val1)) = down(address_of(p2)(ptr_val2))

End Cpp_pointer

% 8.3.2 Reference
%=================
Cpp_reference : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  r : Var Cpp_Subtype(reference?)

  % Alignment
  %-----------
  align_ref(r) : posnat

  % Value Range
  %-------------
  address_range(r) : PRED[Memory_Address]

  % 8.3.2 pt 3 "It is unspecified whether or not a reference requires storage"
  Reference_Type : Datatype
  Begin
    % References stored in an allocated reference variable
    alloc (ref_variable : Address) : alloc?

    % References that are not allocated
    nonalloc (target : Memory_Address) : nonalloc?

  End Reference_Type

  range_reference(r) : PRED[Reference_Type] =
    Lambda (rt : Reference_Type) :
      Cases rt Of
        alloc(ref_var) : divides(align_ref(r), offset(ref_var)),
        nonalloc(target) : address_range(r)(target)
      EndCases

  alloc_range(r)(rt : Reference_Type) : bool =
    range_reference(r)(rt) And alloc?(rt)


  % To do: alignment of references
  % 3.9 pt 5 objects are allocated at an address that meets the alignment
  % requirements
  % reference_to_aligned_object : Axiom
```

```
  % Forall (r : Reference, addr : (address_range(r))) :
  % divides?(alignment(typ(r)), offset(addr))

  % (Un-)interpreted Datatype
  %-------------------------------

  dt_ref_exists : Axiom
    Exists (dt : (pod_data_type?[(alloc_range(r))])) : true

  % proof status :-)
  % dt_reference_TCC1 : TCC Obligation

  dt_reference(r) : (pod_data_type?[(alloc_range(r))])

End Cpp_reference

% 9 Class / Union
%================
Cpp_class : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  c : Var Cpp_Subtype(class?)

  % Alignment
  %-----------
  align_class(c) : posnat

  % Uninterpreted Datatype
  %------------------------
  % We define no semantic domain for classes and unions

  % proof status :-)
  % uidt_class_TCC1 : TCC Obligation

  uidt_class(c) : {uidt : (uninterpreted_data_type?) | size(uidt) > 0}

End Cpp_class

Cpp_union : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  u : Var Cpp_Subtype(union?)

  % Alignment
  %-----------
  align_union(u) : posnat

  % Uninterpreted Datatype
```

```
  %-----------------------
  % We define no semantic domain for classes and unions

  % proof status :-)
  % uidt_union_TCC1 : TCC Obligation

  uidt_union(u) : {uidt : (uninterpreted_data_type?) | size(uidt) > 0}

End Cpp_union


% 8.3.4 Array
%=============
Cpp_array : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  a : Var Cpp_Subtype(array?)

  % Alignment
  %-----------
  align_arr(a) : posnat

  % Uninterpreted Datatype
  %-----------------------
  % We define no semantic domain for arrays

  % proof status :-)
  % uidt_array_TCC1 : TCC Obligation

  uidt_array(a) : (uninterpreted_data_type?)

  % 8.3.4 pt 6 subscript E1[E2] = *((E1) + (E2))

End Cpp_array


% 8.3.5 Functions
%=================
Cpp_function : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  % Functions
  %-----------

  % 8.3.5 pt 2 variable number of arguments f(const char*, [], ...)
  % To do :
  % handle in semantics compiler by creating and passing var args object
```

```
  % 8.3.6
  % We handle default arguments in the semantics compiler by generating
  % appropriate call stubs

  % To do:
  % - pointer to function
  % - recursive functions

  % proof status :-)
  % uidt_function_TCC1 : TCC Obligation

  uidt_function(f : Cpp_Subtype(function?)) : (uninterpreted_data_type?) =
    (# size := 0, valid? := Lambda (l : list[Byte], a : Address) : False #)

  % Virtual Functions
  %--------------------
  c : Var Cpp_Subtype(class?)

  Vtable : Type+

  vtable(c) : Vtable

  vtable_idt_exists : Axiom
    Exists (idt : (interpreted_data_type?[(singleton(vtable(c)))])) : true

  % proof status :-)
  % dt_vtable_TCC1 : TCC Obligation

  dt_vtable(c) : (interpreted_data_type?[(singleton(vtable(c)))])

End Cpp_function


% 9.6 Bit Fields
%================
%
% We do not consider bit fields to directly participate in
% expressions. The integral promotion (§ 4.5 pt 3) and integral
% conversion (§ 4.7 pt 2 and pt 3) rules for bit fields allow bit fields
% to be regarded as signed or unsigned integers for the purpose of
% arithmetic operations.

Cpp_bitfield : Theory
Begin

  Importing Cpp_enum

  b : Var Cpp_Subtype(bitfield?)

  % Alignment
  %-----------
  align_bitfield(b) : posnat
```

```
% 9.6 pt 1 "Alignment of bit-fields is implementation-defined."

% proof status :-)
% range_bitfield_TCC1 : TCC Obligation

% Value Range
%-------------
% 9.6 pt 3 "It is implementation defined whether a plain [...] char, short,
% int or long bit-field is signed or unsigned."
%
% We take this to mean that bit fields taking viewer bits than the value
% represantation of the type can be signed or unsigned. Thus, we can define
% range(b) to be a subset of range(typ(b)).
range_bitfield(b) : non_empty_finite_set[int]

% proof status :-)
% bitfield_range_underlying_TCC1 : TCC Obligation

% proof status :-)
% bitfield_range_underlying_TCC2 : TCC Obligation

% proof status :-)
% bitfield_range_underlying_TCC3 : TCC Obligation

bitfield_range_underlying : Axiom
  If enum?(typ(b)) Then
    subset?(range_bitfield(b), range_enum(typ(b)))
  Else
    subset?(range_bitfield(b), range_integral(typ(b)))
  Endif

% (Un-)interpreted Datatype
%-------------------------

dt_bitfield_exists : Axiom
  Exists (idt : (interpreted_data_type?[(range_bitfield(b))])) : True

% proof status :-)
% dt_bitfield_TCC1 : TCC Obligation

dt_bitfield(b) : (interpreted_data_type?[(range_bitfield(b))])

% the bitfield is contained in the allocation unit
bitfield_bits : Axiom
  bit_offset(b) + bit_size(b) <= size(uidt(dt_bitfield(b))) * bits_per_byte

% To do:
% - bit_masked(mask, n)
% - mask(to_byte(dt))
% - write_data(with_mask)
% - axioms asserting that non-bitfield_masks are full
```

```
  % only bits in the bitfield are masked
  bitfield_mask : Axiom
    Forall (n : nat, data : (range_bitfield(b)), a : Address) :
      n < size(uidt(dt_bitfield(b))) * bits_per_byte And
      bit_set?(n, to_mask(dt_bitfield(b))(data, a))
   Implies
     (bit_offset(b) <= n And n < bit_offset(b) + bit_size(b))

End Cpp_bitfield


% 8.3.3 Pointer to Member
%=========================
Cpp_pointer_to_member : Theory
Begin

  Importing Cpp_Deep_Type_Wellformedness

  p : Var Cpp_Subtype(pointer_to_member?)

  % Alignment
  %------------
  align_pointer_to_member(p) : posnat

  % Value Range
  %-------------
  % The offsets of all class members of that type. Emptyset if no such member
  % is contained in the class. The pointer can at most be initialised with a
  % member of a base class but not with a member of a derived class.
  range_ptm(p) : finite_set[nat]

  % To do:
  % class members provided we could define an Axiom along the lines:
  % range(p) = {n : nat | Exists (m : member(typ(p))) :
  % member_typ(p) = typ(p)(m) Implies n = offset(m)}

  % (Un-)interpreted Datatype
  %-------------------------

  dt_pointer_to_member_exists : Axiom
    Exists (dt : (pod_data_type?[(range_ptm(p))])) : True

  % proof status :-)
  % dt_ptm_TCC1 : TCC Obligation

  dt_ptm(p) : (pod_data_type?[(range_ptm(p))])

End Cpp_pointer_to_member


Cpp_const_volatile[Data : Type] : Theory
```

```
Begin

  Assuming

    Importing Cpp_Deep_Type_Wellformedness

    idt_exists : Assumption
      Exists (idt : (interpreted_data_type?[Data])) : True

  EndAssuming

  % Volatile Datatypes
  %=====================
  % 3.9.3 pt 1 "The cv-qualified or cv-unqualified versions of a type ...
  % shall have the same representation and alignment requirements"
  %
  % In the present specification the behaviour of all write accesses is that
  % of volatile accesses.
  %
  % The approach currently under discussion is to add a store-buffer like
  % unit --- the compiler layer for volatile accesses --- as an additional
  % memory layer. Non volatile writes may be buffered in this layer. Because
  % of 3.9.3 pt 1 non volatile reads of volatile written data should succeed
  % and return the same value. Therefore, we "have to (???)" use hidden bits
  % to signal a volatile access to the store buffer.
  %
  % Because hidden bits are not architectually justified, we provide two
  % models:
  %
  % - Model 1: which cannot distinguish volatile from non-volatile accesses
  % and which needs no hidden bits, and,
  % - Model 2: in which volatile accesses can be distinguished from
  % non-volatile accesses using hidden bits

  % proof status :-)
  % dt_volatile_TCC1 : TCC Obligation

  dt_volatile(idt : (interpreted_data_type?[Data])) :
    (interpreted_data_type?[Data])

  dt_volatile_size : Axiom
    Forall (idt : (interpreted_data_type?[Data])) :
      size(uidt(dt_volatile(idt))) = size(uidt(idt))

  dt_volatile_pod : Axiom
    Forall (idt : (pod_data_type?[Data])) :
      pod_data_type?[Data](dt_volatile(idt))

  dt_volatile_spec_model_1 : Axiom
    Forall (idt : (interpreted_data_type?[Data])) :
      dt_volatile(idt) = idt
```

```
  %>> MV: The side effect cannot change the object representation from
  % dt(volatile(int)) to dt(int) because the type information is
  % lost at this level. We can only add or remove hidden bits
  % that are set / reset in dt(volatile(int)) and not in dt(int)
  % and which do not participate in the value representation.
  dt_volatile_spec_model_2 : Axiom
    Forall (idt : (interpreted_data_type?[Data])) :
      dt_volatile(idt) /= idt And bits_per_byte > min_bits_per_byte

  %>> shouldn't we instead use something like:
  %>> - member(volatile_bit, hidden_bits_per_byte) and
  %>> - bits_per_byte = min_bits_per_byte + card(hidden_bits_per_byte)

  % Const Datatypes
  %==================

  dt_const(idt : (interpreted_data_type?[Data])) :
    (interpreted_data_type?[Data])

  dt_const_size : Axiom
    Forall (idt : (interpreted_data_type?[Data])) :
      size(uidt(dt_const(idt))) = size(uidt(idt))

  dt_const_pod : Axiom
    Forall (idt : (pod_data_type?[Data])) :
      pod_data_type?[Data](dt_const(idt))

  dt_const_spec_model_1 : Axiom
    Forall (idt : (interpreted_data_type?[Data])) :
      dt_const(idt) = idt

  dt_const_spec_model_2 : Axiom
    Forall (idt : (interpreted_data_type?[Data])) :
      dt_const(idt) /= idt And bits_per_byte > min_bits_per_byte

End Cpp_const_volatile


Cpp_Types : Theory
Begin

  Importing Cpp_class, Cpp_union, Min_Max_Extension,
            Cpp_function, Cpp_array, Cpp_bitfield, Cpp_pointer_to_member,
            Cpp_reference, Cpp_pointer, Cpp_const_volatile

  % Const Volatile Qualifiers
  % 3.9.3 pt 1 "The cv-qualified or cv-unqualified versions of a type ...
  % shall have the same representation and alignment requirements"

  % Alignment (3.9 pt 5)
  %======================
```

```
% proof status :-)
% alignment_TCC1 : TCC Obligation

% proof status :-)
% alignment_TCC2 : TCC Obligation

% proof status :-)
% alignment_TCC3 : TCC Obligation

% proof status :-)
% alignment_TCC4 : TCC Obligation

% proof status :-)
% alignment_TCC5 : TCC Obligation

% proof status :-)
% alignment_TCC6 : TCC Obligation

% proof status :-)
% alignment_TCC7 : TCC Obligation

% proof status :-)
% alignment_TCC8 : TCC Obligation

% proof status :-)
% alignment_TCC9 : TCC Obligation

% proof status :-)
% alignment_TCC10 : TCC Obligation

% proof status :-)
% alignment_TCC11 : TCC Obligation

% proof status :-)
% alignment_TCC12 : TCC Obligation

% proof status :-)
% alignment_TCC13 : TCC Obligation

alignment(typ : Cpp_Type) : Recursive nat =
  Cases typ Of
    uchar : align_uchar,
    schar : align_schar,
    char : align_char,
    short : align_short,
    int : align_int,
    long : align_long,
    longlong : align_longlong,
    ushort : align_ushort,
    uint : align_uint,
    ulong : align_ulong,
    ulonglong : align_ulonglong,
```

```
      wchar_t : align_wchar_t,
      bool : align_bool,
      float : align_float,
      double : align_double,
      longdouble : align_longdouble,
      void : 0,

      array_type(t, s) : align_arr(typ),
      function_type(r, a, v) : 0,
      pointer(t) : align_ptr(typ),
      reference(t) : align_ref(typ),
      class(n, a) : align_class(typ),
      union(n) : align_union(typ),
      bitfield(t, bo, s) : align_bitfield(typ),
      enum(n, t, m, c) : align_enum(typ),
      pointer_to_member(t, m) : align_pointer_to_member(typ),

      const (t) : alignment(t),
      volatile (t) : alignment(t)
    EndCases
  Measure typ By <<

  % proof status :-)
  % pointer_to_aligned_object_TCC1 : TCC Obligation

  % proof status :-)
  % pointer_to_aligned_object_TCC2 : TCC Obligation

  % proof status :-)
  % pointer_to_aligned_object_TCC3 : TCC Obligation

  % Pointer
  %----------
  % 3.9 pt 5 Objects are allocated at an address that meets the alignment
  % requirements
  pointer_to_aligned_object : Axiom
    Forall (p : Cpp_Subtype(cv(pointer?)), ptr_val : (range_pointer(cv_base(p)))) :
      not_null?(p)(ptr_val) Implies
        divides(alignment(typ(cv_base(p))), offset(down(address_of(p)(ptr_val))))

  % proof status :-)
  % align_array_of_type_TCC1 : TCC Obligation

  % proof status :-)
  % align_array_of_type_TCC2 : TCC Obligation

  % Arrays
  %---------
  align_array_of_type : Axiom
    Forall (a : Cpp_Subtype(array?)) :
      divides(alignment(a), alignment(typ(a)))
```

```
  % Small Super Type for Integral C++ Types
  %=========================================

  % proof status :-)
  % range_TCC1 : TCC Obligation

  % proof status :-)
  % range_TCC2 : TCC Obligation

  % proof status :-)
  % range_TCC3 : TCC Obligation

  % proof status :-)
  % range_TCC4 : TCC Obligation

  % proof status :-)
  % range_TCC5 : TCC Obligation

  % proof status :-)
  % range_TCC6 : TCC Obligation

  % proof status :-)
  % range_TCC7 : TCC Obligation

  % this range does not include the not allocated references
  range(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) : Recursive
      non_empty_finite_set[int] =
    Cases typ Of
      enum(n, t, m, c) : range_enum(typ),
      const(t) : range(t),
      volatile(t) : range(t)
    Else
      range_integral(typ)
    EndCases
   Measure typ By <<

  % proof status :-)
  % range_cv_base_TCC1 : TCC Obligation

  % proof status :-)
  range_cv_base : Lemma
    Forall (typ : Cpp_Subtype(cv(non_bool_integral_enum?))) :
      range(typ) = range(cv_base(typ))

  % (Un-)interpreted Datatype
  %===========================
  Importing Interpreted_Data_Lift

% To do:
% - we probably need hidden bits to detect const modification through
% an alias and to distinguish volatile from non-volatile accesses
% - Hendrik's idea: use separate datatype for t and const(t);
```

```
% semantics compiler ensures no assignments with const(t)

  % proof status :-)
  % dt_non_bool_integral_enum_TCC1 : TCC Obligation

  % proof status :-)
  % dt_non_bool_integral_enum_TCC2 : TCC Obligation

  % proof status :-)
  % dt_non_bool_integral_enum_TCC3 : TCC Obligation

  % proof status :-)
  % dt_non_bool_integral_enum_TCC4 : TCC Obligation

  % Interpreted Datatype
  %-----------------------------
  dt_non_bool_integral_enum(typ : Cpp_Subtype(non_bool_integral_enum?)) :
      (interpreted_data_type?[(range(typ))]) =
    Cases typ Of
      enum(n, t, m, c) : dt_enum(typ)
    Else
      dt_integral(typ)
    EndCases

  % proof status :-)
  dt_non_bool_integral_enum_pod : Judgement
    dt_non_bool_integral_enum(typ : Cpp_Subtype(non_bool_integral_enum?))
      Has_Type (pod_data_type?[(range(typ))])

  % proof status :-)
  % dt_TCC1 : TCC Obligation

  % proof status :-)
  % dt_TCC2 : TCC Obligation

  % proof status :-)
  % dt_TCC3 : TCC Obligation

  % proof status :-)
  % dt_TCC4 : TCC Obligation

  % proof status :-)
  % dt_TCC5 : TCC Obligation

  % Generic Datatype Model
  %=========================
  dt(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) : Recursive
      (interpreted_data_type?[(range(typ))]) =
    Cases typ Of
      const(t) : dt_const(dt(t)),
      volatile(t) : dt_volatile(dt(t))
    Else
```

```
    dt_non_bool_integral_enum(typ)
  EndCases
 Measure typ by <<

% proof status :-)
dt_pod : Judgement
  dt(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) Has_Type
    (pod_data_type?[(range(typ))])

% proof status :-)
% dt_cv_pointer_TCC1 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC2 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC3 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC4 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC5 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC6 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC7 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC8 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC9 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC10 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC11 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC12 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC13 : TCC Obligation

% proof status :-)
% dt_cv_pointer_TCC14 : TCC Obligation

% proof status :-)
```

```
  % dt_cv_pointer_TCC15 : TCC Obligation

  dt_cv_pointer(typ : Cpp_Subtype(cv(pointer?))) : Recursive
      (interpreted_data_type?[(range_pointer(cv_base(typ)))]) =
    Cases typ Of
      const(t) : dt_const(dt_cv_pointer(t)),
      volatile(t) : dt_volatile(dt_cv_pointer(t))
    Else
      dt_pointer(typ)
    EndCases
   Measure typ by <<

  % proof status :-)
  % dt_cv_bitfield_TCC1 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC2 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC3 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC4 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC5 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC6 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC7 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC8 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC9 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC10 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC11 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC12 : TCC Obligation

  % proof status :-)
  % dt_cv_bitfield_TCC13 : TCC Obligation

  % proof status :-)
```

```
% dt_cv_bitfield_TCC14 : TCC Obligation

% proof status :-)
% dt_cv_bitfield_TCC15 : TCC Obligation

dt_cv_bitfield(typ : Cpp_Subtype(cv(bitfield?))) : Recursive
    (interpreted_data_type?[(range_bitfield(cv_base(typ)))]) =
  Cases typ Of
    const(t) : dt_const(dt_cv_bitfield(t)),
    volatile(t) : dt_volatile(dt_cv_bitfield(t))
  Else
    dt_bitfield(typ)
  EndCases
 Measure typ by <<

% proof status :-)
% dt_cv_ptm_TCC1 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC2 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC3 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC4 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC5 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC6 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC7 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC8 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC9 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC10 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC11 : TCC Obligation

% proof status :-)
% dt_cv_ptm_TCC12 : TCC Obligation

% proof status :-)
```

```
  % dt_cv_ptm_TCC13 : TCC Obligation

  % proof status :-)
  % dt_cv_ptm_TCC14 : TCC Obligation

  % proof status :-)
  % dt_cv_ptm_TCC15 : TCC Obligation

  dt_cv_ptm(typ : Cpp_Subtype(cv(pointer_to_member?))) : Recursive
      (interpreted_data_type?[(range_ptm(cv_base(typ)))]) =
    Cases typ Of
      const(t) : dt_const(dt_cv_ptm(t)),
      volatile(t) : dt_volatile(dt_cv_ptm(t))
    Else
      dt_ptm(typ)
    EndCases
   Measure typ by <<

  % proof status :-)
  % dt_cv_float_TCC1 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC2 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC3 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC4 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC5 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC6 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC7 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC8 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC9 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC10 : TCC Obligation

  % proof status :-)
  % dt_cv_float_TCC11 : TCC Obligation

  % proof status :-)
```

```
% dt_cv_float_TCC12 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC13 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC14 : TCC Obligation

% proof status :-)
% dt_cv_float_TCC15 : TCC Obligation

dt_cv_float(typ : Cpp_Subtype(cv(floating_point?))) : Recursive
    (interpreted_data_type?[(range_floating_point(cv_base(typ)))]) =
  Cases typ Of
    const(t) : dt_const(dt_cv_float(t)),
    volatile(t) : dt_volatile(dt_cv_float(t))
  Else
    dt_floating_point(typ)
  EndCases
 Measure typ by <<

% Dt is pod judgements
%-----------------------
% To do (if required)

% proof status :-)
% uidt_TCC1 : TCC Obligation

% proof status :-)
% uidt_TCC2 : TCC Obligation

% proof status :-)
% uidt_TCC3 : TCC Obligation

% proof status :-)
% uidt_TCC4 : TCC Obligation

% proof status :-)
% uidt_TCC5 : TCC Obligation

% proof status :-)
% uidt_TCC6 : TCC Obligation

% proof status :-)
% uidt_TCC7 : TCC Obligation

% proof status :-)
% uidt_TCC8 : TCC Obligation

% proof status :-)
% uidt_TCC9 : TCC Obligation
```

```
% proof status :-)
% uidt_TCC10 : TCC Obligation

% proof status :-)
% uidt_TCC11 : TCC Obligation

% proof status :-)
% uidt_TCC12 : TCC Obligation

% proof status :-)
% uidt_TCC13 : TCC Obligation

% proof status :-)
% uidt_TCC14 : TCC Obligation

% proof status :-)
% uidt_TCC15 : TCC Obligation

% proof status :-)
% uidt_TCC16 : TCC Obligation

% proof status :-)
% uidt_TCC17 : TCC Obligation

% proof status :-)
% uidt_TCC18 : TCC Obligation

% proof status :-)
% uidt_TCC19 : TCC Obligation

% proof status :-)
% uidt_TCC20 : TCC Obligation

% proof status :-)
% uidt_TCC21 : TCC Obligation

% proof status :-)
% uidt_TCC22 : TCC Obligation

% proof status :-)
% uidt_TCC23 : TCC Obligation

% proof status :-)
% uidt_TCC24 : TCC Obligation

% proof status :-)
% uidt_TCC25 : TCC Obligation

% proof status :-)
% uidt_TCC26 : TCC Obligation

% proof status :-)
```

```
% uidt_TCC27 : TCC Obligation

% proof status :-)
% uidt_TCC28 : TCC Obligation

% proof status :-)
% uidt_TCC29 : TCC Obligation

% proof status :-)
% uidt_TCC30 : TCC Obligation

% proof status :-)
% uidt_TCC31 : TCC Obligation

% proof status :-)
% uidt_TCC32 : TCC Obligation

% proof status :-)
% uidt_TCC33 : TCC Obligation

% proof status :-)
% uidt_TCC34 : TCC Obligation

% proof status :-)
% uidt_TCC35 : TCC Obligation

% proof status :-)
% uidt_TCC36 : TCC Obligation

% proof status :-)
% uidt_TCC37 : TCC Obligation

% proof status :-)
% uidt_TCC38 : TCC Obligation

% proof status :-)
% uidt_TCC39 : TCC Obligation

% proof status :-)
% uidt_TCC40 : TCC Obligation

% proof status :-)
% uidt_TCC41 : TCC Obligation

% Uninterpreted Datatype
%-----------------------
uidt(typ : Cpp_Type) : Recursive (uninterpreted_data_type?) =
  Cases typ Of
    void : uidt_void,
    array_type(t, s) : uidt_array(typ),
    function_type(r, a, v) : uidt_function(typ),
    class(n, a) : uidt_class(typ),
```

```
    union(n) : uidt_union(typ),

    % interpreted types
    pointer(t) : uidt(dt_cv_pointer(typ)),
    reference(t) : uidt(dt_reference(typ)),
    pointer_to_member(c, t) : uidt(dt_cv_ptm(typ)),
    bitfield(t, bo, s) : uidt(dt_cv_bitfield(typ)),

    bool : uidt(dt_bool),

    % qualifiers
    const(t) :
      Cases t Of
        bool : uidt(dt_bool),
        pointer(t1) : uidt(dt_cv_pointer(typ)),
        pointer_to_member(c, t1) : uidt(dt_cv_ptm(typ)),
        bitfield(t1, bo, s) : uidt(dt_cv_bitfield(typ)),
        volatile(t1) : uidt(t)
      Else
        Cond
          floating_point?(t) -> uidt(dt_cv_float(typ)),
          non_bool_integral_enum?(t) -> uidt(dt(typ))
        EndCond
      EndCases,

    volatile(t) :
      Cases t Of
        bool : uidt(dt_bool),
        pointer(t1) : uidt(dt_cv_pointer(typ)),
        pointer_to_member(c, t1) : uidt(dt_cv_ptm(typ)),
        bitfield(t1, bo, s) : uidt(dt_cv_bitfield(typ))
      Else
        Cond
          floating_point?(t) -> uidt(dt_cv_float(typ)),
          non_bool_integral_enum?(t) -> uidt(dt(typ))
        EndCond
      EndCases

  Else
    Cond
      floating_point?(typ) -> uidt(dt_cv_float(typ)),
      non_bool_integral_enum?(typ) -> uidt(dt(typ))
    EndCond
  EndCases
 Measure typ by <<

% Array
%-------
% 8.3.4 pt 1 contiguously allocated set of N objects
array_size : Axiom
  Forall (a : Cpp_Subtype(array?)) :
    size(uidt(a)) = size(a) * size(uidt(typ(a)))
```

```
% Size Of (3.9 pt 4)
%-----------------------
size_of(typ : Cpp_Type) : nat = size(uidt(typ))

% Modulo Arithmetic / Floating Point Values
%-----------------------------------------
% arithmetic types (except bool), enums and bitfields may follow the rules
% of arithmetic modulo 2^n where n is the number of bits in the value
% representation. Floating point values may have to be adjusted to a
% representable higher or lower value. Pointers effectively wrap around,
% although the standard leaves this behaviour undefined:
% (e.g., char * p = &0xffffffff p++ = 0).
%
% We therefore allow value modifications for any interpreted type.
%
% Completing the specification of the below underspecified function
% should allow us to add this functionality. For example
% mod(typ : cv(unsigned))(n) = n mod 2^max_value_bits(typ)

mod(typ : Cpp_Subtype(cv(non_bool_integral_enum?)))(n : int) : int

mod_range : Axiom
  Forall (typ : Cpp_Subtype(cv(non_bool_integral_enum?)), n : (range(typ))) :
    mod(typ)(n) = n

% proof status :-)
% mod_float_TCC1 : TCC Obligation

mod_float(typ : Cpp_Subtype(cv(floating_point?)))(n : extended_real) :
  extended_real

mod_float_range : Axiom
  Forall (typ : Cpp_Subtype(cv(floating_point?)),
          n : (range_floating_point(cv_base(typ)))) :
    mod_float(typ)(n) = n

% proof status :-)
% value_bitmask_TCC1 : TCC Obligation

% bits in value representation
value_bitmask(typ : Cpp_Subtype(cv(non_bool_integral_enum?))) : Recursive
    list[Byte] =
  Cases typ Of
    char : value_bitmask_char,
    schar : value_bitmask_schar,
    uchar : value_bitmask_uchar,
    short : value_bitmask_short,
    ushort : value_bitmask_ushort,
    int : value_bitmask_int,
    uint : value_bitmask_uint,
    long : value_bitmask_long,
```

```
      ulong : value_bitmask_ulong,
      longlong : value_bitmask_longlong,
      ulonglong : value_bitmask_ulonglong,
      wchar_t : value_bitmask_wchar_t,
      enum(n,t,m,c) : value_bitmask_enum(typ),
      const(t) : value_bitmask(t),
      volatile(t) : value_bitmask(t)
    EndCases
  Measure typ by <<

  Auto_Rewrite- Cpp_Type?

End Cpp_Types
```

B.33 vfiasco-prelude.pvs

```
Unit : Datatype
% the unit type (aka semantic of void)
Begin
  unit : unit?
End Unit

More_Divides : Theory
Begin

  n,m : Var nat

  % proof status :-)
  expt_divides : Lemma Forall(a : posnat) : a > 1 And n <= m Implies
    divides(expt(a, n), expt(a, m))

  % proof status :-)
  divides_expt_gt : Lemma Forall(i, j : nat) :
    j <= i And divides(expt(2, i), n) Implies divides(expt(2, j), n)

End More_Divides


Expt_Lemmas : Theory
Begin
  m, n: VAR nat
  n0x : VAR nzreal
  gt1x, gt1y: VAR {r: posreal | r > 1}

  % proof status :-)
  both_sides_expt_gt1_le: LEMMA
    m <= n IFF expt(gt1x, m) <= expt(gt1x, n)

  % proof status :-)
  expt_plus : Lemma Forall(n0x : nzreal, m, n : nat) :
    expt(n0x, m) * expt(n0x, n) = expt(n0x, m + n)
```

```
End Expt_Lemmas


Posnat_induction : Theory
Begin

  p: VAR pred[posnat]
  i, j : VAR posnat

  % proof status :-)
  posnat_induction : Lemma
    (p(1) And (Forall j: p(j) Implies p(j+1)))
        Implies (Forall i: p(i))

End Posnat_induction


Number_Props : Theory
Begin
  Importing More_Divides, Expt_Lemmas


  % proof status :-)
  number_split : Lemma Forall(x : int, a : posnat) :
    x = a * ndiv(x, a) + rem(a)(x)

  % proof status :-)
  number_split_less : Lemma Forall(a, b, p, q : nat) :
    a < p Implies
      (a + b * p < p * q IFF b < q)

  % proof status :-)
  bit_split_less : Lemma Forall(q, r, j, k : nat) :
    r < expt(2, k) Implies
      (q * expt(2, k) + r < expt(2, k + j) IFF
        q < expt(2, j))

  % proof status :-)
  rem_def_pos : Lemma Forall(b : posnat, x : nat) :
     Forall(r : mod(b)): rem(b)(x) = r IFF Exists(q : nat): x = b * q + r

  % proof status :-)
  rem_rem : Lemma Forall(a, b : posnat, x : int) :
    divides(a, b) Implies
      rem(a)(rem(b)(x)) = rem(a)(x)

  % proof status :-)
  divides_times : Lemma Forall(i : posnat, j, k : int) :
    divides(i * j, k) IFF divides(i, k) And divides(j, ndiv(k, i))

  % proof status :-)
```

```
ndiv_floor : Lemma Forall(a : int, b : posnat) : ndiv(a,b) = floor(a/b)

% proof status :-)
ndiv_self : Lemma Forall(a : posnat) : ndiv(a,a) = 1

% proof status :-)
ndiv_plus_1 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, a) Implies
    ndiv(a + b, c) = ndiv(a,c) + ndiv(b,c)

% proof status :-)
ndiv_plus_2 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, b) Implies
    ndiv(a + b, c) = ndiv(a,c) + ndiv(b,c)

% proof status :-)
ndiv_plus_mod : Lemma Forall(a : int, c : posnat, b : mod(c)) :
  divides(c, a) Implies
    ndiv(a + b, c) = ndiv(a, c)

% proof status :-)
ndiv_plus_mod_2 : Lemma Forall(a : int, c : posnat, b1, b2 : mod(c)) :
  divides(c, a) Implies
    ndiv(a + b1, c) = ndiv(a + b2, c)

% proof status :-)
ndiv_minus_2 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, b) Implies
    ndiv(a - b, c) = ndiv(a,c) - ndiv(b,c)

% ndiv_minus_1 is much harder, because it might
% ndiv(a - b, c) = ndiv(a,c) - ndiv(b,c) -1
% if rem(c)(b) > 0 or might it not??

% proof status :-)
ndiv_times_divident_1 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, a) Implies
    ndiv(a * b, c) = ndiv(a,c) * b

% proof status :-)
ndiv_times_divident_2 : Lemma Forall(a, b : int, c : posnat) :
  divides(c, b) Implies
    ndiv(a * b, c) = a * ndiv(b,c)

% proof status :-)
rem_prod_3 : Lemma Forall(b1, b2 : posnat, x : int) :
   rem(b1 * b2)(x) = rem(b1)(x) + rem(b2)(ndiv(x, b1)) * b1

% proof status :-)
ndiv_times_2 : Lemma Forall(a : int, b, c : posnat) :
  ndiv(a, b * c) = ndiv(ndiv(a, b), c)
```

```
% proof status :-)
both_sides_ndiv_lt1 : Lemma Forall(a, b : int, c : posnat) :
  a < b And divides(c, b) Implies
    ndiv(a,c) < ndiv(b,c)

% proof status :-)
rem_ndiv : Lemma Forall(a : posnat, b : int, c : posnat) :
  rem(a)(ndiv(b, c)) = ndiv(rem(c * a)(b), c)

% proof status :-)
mod_rem : Lemma Forall(i : int, b : posnat) :
  mod(i, b) = rem(b)(i)

% proof status :-)
% ndiv_rem_divisible_TCC1 : TCC Obligation

% proof status :-)
ndiv_rem_divisible : Lemma Forall(a, c : posnat, b : int) :
  divides(c, a) And c <= a Implies
    ndiv(rem(a)(b), c) = rem(ndiv(a, c))(ndiv(b,c))

% proof status :-)
expt_2_8 : Lemma expt(2, 8) = 256

% proof status :-)
ndiv_0 : Lemma Forall(b : posnat) : ndiv(0, b) = 0

% proof status :-)
ndiv_1 : Lemma Forall(i : int) : ndiv(i, 1) = i

% proof status :-)
ndiv_bigger : Lemma Forall(n : nat, m : posnat) :
  n < m Implies ndiv(n, m) = 0

% proof status :-)
ndiv_reduce : Lemma Forall(a : int, b : posnat, c : int, d : posnat) :
  a * d = b * c Implies
    ndiv(a,b) = ndiv(c,d)

% proof status :-)
% ndiv_expt_expt_TCC1 : TCC Obligation

% proof status :-)
ndiv_expt_expt : Lemma Forall(a : posnat, n, m : nat) :
 a > 1 Implies
   ndiv(expt(a, n), expt(a, m)) =
     IF n < m Then 0
     Else expt(a, n - m)
     Endif

% Floor : x - rem(x / b)
%=======================
```

```
  floor(b : posnat)(x : int) : int =
    x − rem(b)(x)

  % proof status :−)
  % min_plus1_TCC1 : TCC Obligation

  % proof status :−)
  min_plus1 : Lemma
    Forall (P : (nonempty?[nat])) :
      Not P(0) Implies
        min(P) = min(Lambda (n : nat) : P(n + 1)) + 1

  % proof status :−)
  min_member : Lemma
    Forall (P : (nonempty?[nat]), n : nat) :
      min(P) = n Implies P(n)

End Number_Props


Extended_Real : Theory
Begin

  % The purpose of this theory is to define arithmetic on real numbers
  % that can possibly contain infinity values (as e.g., in x / oo = 0),
  % not−a−thing values (e.g., x / 0 = NAT) and exception values (not
  % currently supported, i.e., no exception will be generated).

  extended_real? : PRED[number]

  extended_real_superset_of_number_fields : Axiom
    Forall (n : number_field) : extended_real?(n)

  extended_real : Type = (extended_real?)

  extended_real_number : Judgement
    extended_real Subtype_Of number

  % proof status :−)
  number_field_extended : Judgement
    number_field Subtype_Of extended_real

  % extended real ops
  %−−−−−−−−−−−−−−−−−−−
  x, y : Var extended_real

  % proof status :−)
  % er_plus_TCC1 : TCC Obligation

  % proof status :−)
  % er_neg_TCC1 : TCC Obligation
```

```
er_plus(x, y) : extended_real
er_minus(x, y) : extended_real
er_times(x, y) : extended_real
er_div(x, y) : extended_real
er_neg(x) : extended_real

% on number fields these ops behave like the number field ops
%-----------------------------------------------------------
a, b : Var number_field

er_plus_ax : Axiom
  er_plus(a, b) = number_fields.+(a, b)

er_minus_ax : Axiom
  er_minus(a, b) = number_fields.-(a, b)

er_times_ax : Axiom
  er_times(a, b) = number_fields.*(a, b)

er_div_ax : Axiom
  b /= 0 Implies er_div(a, b) = number_fields./(a, b)

er_neg_ax : Axiom
  er_neg(a) = number_fields.-(a)

% Judgements:
%------------

% number_field
%~~~~~~~~~~~~~
% proof status :-)
er_plus_number_field : Judgement
  er_plus(a, b : number_field) Has_Type number_field

% proof status :-)
er_minus_number_field : Judgement
  er_minus(a, b : number_field) Has_Type number_field

% proof status :-)
er_times_number_field : Judgement
  er_times(a, b : number_field) Has_Type number_field

% proof status :-)
er_div_number_field : Judgement
  er_div(a : number_field, b : nznum) Has_Type number_field

% proof status :-)
er_neg_number_field : Judgement
  er_neg(a : number_field) Has_Type number_field

% real
```

```
%~~~~~~
% proof status :-)
er_plus_real : Judgement
  er_plus(a, b : real) Has_Type real

% proof status :-)
er_minus_real : Judgement
  er_minus(a, b : real) Has_Type real

% proof status :-)
er_times_real : Judgement
  er_times(a, b : real) Has_Type real

% proof status :-)
er_div_real : Judgement
  er_div(a : real, b : nzreal) Has_Type real

% proof status :-)
er_neg_real : Judgement
  er_neg(a : real) Has_Type real

% rational
%~~~~~~~~~~
% proof status :-)
er_plus_rational : Judgement
  er_plus(a, b : rational) Has_Type rational

% proof status :-)
er_minus_rational : Judgement
  er_minus(a, b : rational) Has_Type rational

% proof status :-)
er_times_rational : Judgement
  er_times(a, b : rational) Has_Type rational

% proof status :-)
er_div_rational : Judgement
  er_div(a : rational, b : nzrat) Has_Type rational

% proof status :-)
er_neg_rational : Judgement
  er_neg(a : rational) Has_Type rational

% integers
%~~~~~~~~~~
% proof status :-)
er_plus_integer : Judgement
  er_plus(a, b : int) Has_Type int

% proof status :-)
er_minus_integer : Judgement
  er_minus(a, b : int) Has_Type int
```

```
  % proof status :-)
  er_times_integer : Judgement
    er_times(a, b : int) Has_Type int

  % proof status :-)
  er_neg_integer : Judgement
    er_neg(a : int) Has_Type int

END Extended_Real


More_Sets_Lemmas[T: TYPE]: THEORY
BEGIN
  a, b, c : VAR set[T]

  %% (auto-rewrite
  %% "subset_equal" "union_subset1" "union_subset3"
  %% "union_left" "union_right" "subset_singleton"
  %% "subset_bigger_union_right" "subset_bigger_union_left"
  %% "union_left" "union_right" )

  % proof status :-)
  subset_equal : Lemma
    a = b Implies subset?(a, b)

  % proof status :-)
  union_subset3: LEMMA subset?(a, union(b, a))

  % proof status :-)
  subset_bigger_union_left : Lemma
    subset?(a, b) Implies subset?(a, union(b, c))

  % proof status :-)
  subset_bigger_union_right : Lemma
    subset?(a, c) Implies subset?(a, union(b, c))

  % proof status :-)
  union_empty_left : Lemma
    union(emptyset, a) = a

  % proof status :-)
  union_left : Lemma Forall (e : T) :
    a(e) Implies union(a, b)(e)

  % proof status :-)
  union_right : Lemma Forall (e : T) :
    b(e) Implies union(a, b)(e)

  % proof status :-)
  subset_of_differences : Lemma
    subset?(a, b) Implies subset?(difference(c, b), difference(c, a))
```

```
  % proof status :-)
  difference_disjoint_3: LEMMA disjoint?(difference(b, a), a)

  % proof status :-)
  subset_singleton : Lemma Forall(e : T) :
    subset?(singleton(e), a) = a(e)

  % proof status :-)
  subset_member : Lemma Forall(e : T) :
    a(e) And subset?(a, b) Implies b(e)

  % proof status :-)
  disjoint_symmetric : Lemma
    disjoint?(a, b) = disjoint?(b, a)

  % proof status :-)
  disjoint_mono : Lemma Forall(a1, a2, b1, b2 : set[T]) :
    subset?(a1, a2) And subset?(b1, b2) And
    disjoint?(a2, b2) Implies
      disjoint?(a1, b1)

  % proof status :-)
  disjoint_subset_difference : Lemma
    subset?(a, b) And disjoint?(a, c) Implies
      subset?(a, difference(b, c))

End More_Sets_Lemmas


Set_Lift[Data1, Data2 : Type] : Theory
Begin

  set_lift(P : PRED[Data1], inj : [Data1 -> Data2]) : PRED[Data2] =
    Lambda (d2 : Data2) : Exists (d1 : (P)) : inj(d1) = d2

End Set_Lift


Alignment : Theory
Begin
  Importing Number_Props

  n,m : Var nat

  aligned?(n) : PRED[nat] = { a : nat | divides(expt(2,n), a) }

  aligned(n) : Type = (aligned?(n))

  % proof status :-)
  aligned_zero : Lemma aligned?(n)(0)
```

```
  % proof status :-)
  zero_aligned : Lemma aligned?(0)(n)

  % proof status :-)
  aligned_bigger : Lemma Forall(n1, n2 : nat) :
    n2 <= n1 And aligned?(n1)(m) Implies aligned?(n2)(m)

% aligned_add : Judgement +(a:aligned(n), b:aligned(m))
% HAS_TYPE aligned(min(n,m))

  % proof status :-)
  aligned_plus : Lemma
    Forall(a : aligned(n), b : aligned(m)) : aligned?(min(n,m))(a + b)

  % proof status :-)
  aligned_plus_n : Lemma
    Forall(a, b : aligned(n)) : aligned?(n)(a + b)

  % proof status :-)
  % aligned_minus_TCC1 : TCC Obligation

  % proof status :-)
  aligned_minus : Lemma
    Forall(a : aligned(n), b : aligned(m)) :
        a >= b Implies aligned?(min(n,m))(a - b)

  % proof status :-)
  aligned_mult_expt_2 : Lemma
    Forall(a : aligned(n)) : aligned?(n + m)(a * expt(2, m))

  % proof status :-)
  aligned_rem : Lemma Forall(i : nat) :
    aligned?(n)(m) Implies aligned?(n)(rem(expt(2, i))(m))

  % aligned_minus_rem : Lemma Forall (i : nat) :
  % aligned?(i)(n - rem(expt(2, i))(n))
  % >> (use "rem_def2")(expand "aligned?")(smash)

  % proof status :-)
  aligned_add_below : Lemma Forall (n, m, i, k : nat) :
    i <= k And n < expt(2, k) And aligned?(i)(n) And m < expt(2, i) Implies
      n + m < expt(2, k)

End Alignment


More_List_Props[T : Type] : Theory
Begin

  % proof status :-)
  length_is_cons : Lemma Forall(l : list[T]) :
    length(l) > 0 Implies cons?(l)
```

```
% proof status :-)
cons_length : Lemma
  Forall(l : (cons?[T])) : length(l) > 0

% proof status :-)
% length_cdr_TCC1 : TCC Obligation

% proof status :-)
length_cdr : Lemma Forall(l : list[T]) :
  length(l) > 0 Implies
    length(cdr(l)) = length(l) - 1

% proof status :-)
conslist_induction: Lemma
  Forall(p: [(cons?[T]) -> boolean]):
    (Forall(t : T) : p(cons(t, null))) And
    (Forall(t : T, tail : (cons?[T])): p(tail) Implies p(cons(t, tail)))
      Implies
        Forall (l : (cons?[T])): p(l)

% proof status :-)
% head_TCC1 : TCC Obligation

% proof status :-)
% head_TCC2 : TCC Obligation

% proof status :-)
% head_TCC3 : TCC Obligation

head(l : list[T], n : upto(length(l))) : Recursive list[T] =
  If n = 0 Then
    null
  Else
    cons(car(l), head(cdr(l), n - 1))
  Endif
 Measure n

tail(l : list[T], n : upto(length(l))) : Recursive list[T] =
  If n = 0 Then
    l
  Else
    tail(cdr(l), n - 1)
  Endif
 Measure n

% proof status :-)
head_tail : Lemma Forall (l : list[T], n : nat) :
  n <= length(l) Implies
    append(head(l, n), tail(l, n)) = l

% proof status :-)
```

```
length_head : Lemma Forall (l : list[T], n : nat) :
  n <= length(l) Implies length(head(l, n)) = n

% proof status :-)
length_tail : Lemma Forall (l : list[T], n : nat) :
  n <= length(l) Implies length(tail(l, n)) = length(l) - n

% proof status :-)
% nth_head_TCC1 : TCC Obligation

% proof status :-)
% nth_head_TCC2 : TCC Obligation

% proof status :-)
nth_head : Lemma Forall(l : list[T], n, i : nat) :
  n <= length(l) And i < n Implies
    nth(head(l, n), i) = nth(l, i)

% proof status :-)
% nth_tail_TCC1 : TCC Obligation

% proof status :-)
% nth_tail_TCC2 : TCC Obligation

% proof status :-)
nth_tail : Lemma Forall(l : list[T], n, i : nat) :
  n <= length(l) And i + n < length(l) Implies
    nth(tail(l, n), i) = nth(l, n + i)

% proof status :-)
% car_head_TCC1 : TCC Obligation

% proof status :-)
% car_head_TCC2 : TCC Obligation

% proof status :-)
car_head : Lemma Forall(l : list[T], n : upto(length(l))) :
  n > 0 Implies
    car(head(l, n)) = car(l)

% proof status :-)
cons_tail : Lemma Forall(l : list[T], n : upto(length(l))) :
  n < length(l) Implies
    cons?(tail(l, n))

% proof status :-)
% car_tail_TCC1 : TCC Obligation

% proof status :-)
car_tail : Lemma Forall(l : list[T], n : upto(length(l))) :
  n < length(l) Implies
    car(tail(l, n)) = nth(l, n)
```

```
% proof status :-)
% cdr_tail_TCC1 : TCC Obligation

% proof status :-)
cdr_tail : Lemma Forall(l : list[T], n : upto(length(l))) :
  n < length(l) Implies
    cdr(tail(l, n)) = tail(l, n+1)


% proof status :-)
every_implied : Lemma Forall(l : list[T], P, Q : PRED[T]) :
  (Forall (t : T) : P(t) Implies Q(t)) And
     every(P)(l)
  Implies
    every(Q)(l)

% proof status :-)
every_conjunct_left : Lemma Forall(l : list[T], P, Q : PRED[T]) :
  every(P)(l) And every(Q)(l) Implies every(lambda(t : T) : P(t) And Q(t))(l)

% proof status :-)
every_iff_forall : Lemma Forall (l : list[T], P : PRED[T]) :
  every(P)(l) IFF (Forall (t : T) : member(t, l) Implies P(t))

% proof status :-)
some_iff_exists : Lemma Forall (l : list[T], P : PRED[T]) :
  some(P)(l) IFF (Exists (t : T) : member(t, l) And P(t))


% proof status :-)
% list_extensionality_TCC1 : TCC Obligation

% proof status :-)
list_extensionality : Lemma Forall(l1, l2 : list[T]) :
    l1 = l2 IFF
    (length(l1) = length(l2) AND
     Forall(i : below(length(l1))) : nth(l1,i) = nth(l2,i))


% proof status :-)
% list_remove_TCC1 : TCC Obligation

% proof status :-)
% list_remove_TCC2 : TCC Obligation

list_remove(x : T, l : list[T]) : Recursive list[T] =
  Cases l OF
    null : null,
    cons(car, cdr) :
      IF x = car Then list_remove(x, cdr)
      Else cons(car, list_remove(x, cdr))
```

```
      Endif
  EndCases
Measure length(l)


% proof status :-)
list_remove_commutative : Lemma
  Forall(x, y : T, l : list[T]) :
    list_remove(x, list_remove(y, l)) = list_remove(y, list_remove(x, l))

% proof status :-)
member_list_remove : Lemma
  Forall(x, y : T, l : list[T]) :
    member(x, list_remove(y, l)) =
      IF x = y then false
      Else member(x, l)
      EndIF


% proof status :-)
nth_member : Lemma
  Forall(x : T, l : list[T]) :
    member(x, l) IFF
    Exists(i : below(length(l))) : nth(l, i) = x


% proof status :-)
% flatten_TCC1 : TCC Obligation

flatten(ll : list[list[T]]) : Recursive list[T] =
  Cases ll OF
    null : null,
    cons(car, cdr) : append(car, flatten(cdr))
  EndCases
Measure length(ll)


% proof status :-)
member_append : Lemma
  Forall(x : T, l1, l2 : list[T]) :
    member(x, append(l1, l2)) IFF
      member(x, l1) OR member(x, l2)

% proof status :-)
member_flatten : Lemma
  Forall(x : T, ll : list[list[T]]) :
    member(x, flatten(ll)) IFF
    Exists(l : list[T]) :
      member(l, ll) And member(x, l)


% proof status :-)
```

```
  % map2_TCC1 : TCC Obligation

  map2(f : [T, T -> T])(l1, l2 : list[T]) : Recursive list[T] =
    Cases l1 Of
      null : null,
      cons(h1, t1) :
        Cases l2 Of
          null : null,
          cons(h2, t2) : cons(f(h1, h2), map2(f)(t1, t2))
        EndCases
    EndCases
   Measure length(l1)

  % proof status :-)
  map2_property : Lemma
    Forall (P : PRED[T], f : [T, T -> T], l1, l2 : list[T]) :
      (Forall (t1, t2 : T) : P(f(t1, t2))) Implies
        every(P)(map2(f)(l1, l2))

End More_List_Props


More_List_Props_2[A, B : Type] : Theory
Begin

  % proof status :-)
  member_map : Lemma
    Forall(b : B, f : [A -> B], l : list[A]) :
      member(b, map(f)(l)) IFF
        Exists(a : A) : member(a, l) And f(a) = b

End More_List_Props_2


More_More_List_Props [T : Type+] : Theory
Begin

  % proof status :-)
  list_all_length : Lemma Forall (n : nat) :
    Exists (l : list[T]) : length(l) = n

End More_More_List_Props


Even_More_List_Props [T : Type, S : Type From T] : Theory
Begin

  % proof status :-)
  every_extend : Lemma
    Forall (l : list[S], P : PRED[T]) :
      every(Lambda (t : T) : P(t))(l) Implies every(Lambda (s : S) : P(s))(l)
```

```
End Even_More_List_Props


More_Relations[T : Type] : Theory
Begin
  Importing More_List_Props % for cons_length

  R : Var PRED[[T,T]]

  % Zorn's lemma (I missed it in the prelude)
  % proof status :-)
  zorn_lr : Lemma
    well_founded?(R) Implies
        Not Exists (f : [nat -> T]) : Forall (n : nat) : R(f(n+1), f(n))

End More_Relations


More_Function[Domain, Range : Type] : Theory
% some more stuff on functions
Begin

  f : Var [Domain -> Range]

  % proof status :-)
  % restrict_to_image_TCC1 : TCC Obligation

  restrict_to_image(f) : [Domain -> (image(f, fullset[Domain]))] = f

  % proof status :-)
  surjective_restrict_to_image : Lemma
    surjective?(restrict_to_image(f))

  % proof status :-)
  bijective_restrict_to_image : Lemma
    injective?(f) Implies bijective?(restrict_to_image(f))

End More_Function


Finite_Set_Reduce[T : Type] : Theory
Begin

  Importing More_List_Props % for nth_member

  S : Var finite_set[T]

  % proof status :-)
  % list_of_finite_set_TCC1 : TCC Obligation

  % proof status :-)
  % list_of_finite_set_TCC2 : TCC Obligation
```

```
  list_of_finite_set(S) : Recursive list[T] =
    IF empty?(S) Then null
    Else cons(choose(S), list_of_finite_set(rest(S)))
    Endif
  Measure card(S)

  % proof status :-)
  length_list_of_finite_set : Lemma
    length(list_of_finite_set(S)) = card(S)

  % proof status :-)
  member_list_of_finite_set : Lemma
    Forall(x : T) :
      member(x, S) = member(x, list_of_finite_set(S))

  % proof status :-)
  unique_member_list_of_finite_set : Lemma
    Forall(i, j : below(length(list_of_finite_set(S)))) :
      nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) Implies
        i = j

End Finite_Set_Reduce


Min_Max_Extension[T : Type, S : Type From T, <= : (total_order?[T])] : Theory
Begin

  Importing finite_sets@finite_sets_minmax

  % proof status :-)
  % max_stable_under_extension_TCC1 : TCC Obligation

  % proof status :-)
  max_stable_under_extension : Lemma
    Forall (P : non_empty_finite_set[S]) :
      max[T, <=](extend[T, S, bool, false](P)) = max[S, <=](P)

  % proof status :-)
  min_stable_under_extension : Lemma
    Forall (P : non_empty_finite_set[S]) :
      min[T, <=](extend[T, S, bool, false](P)) = min[S, <=](P)

End Min_Max_Extension
```

C Proof scripts

Contents for Appendix C

C.1 Proofs for Abstract_Read_Write (abstract_data.pvs)

C.1.1 Abstract_Read_Write.read_data_valid_in_mem

Terse proof for `read_data_valid_in_mem`.
`read_data_valid_in_mem`:

{1} ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Memory_struct[State], s: State): OK?(read_data(pm, dt)(addr)(s)) ⊃ valid_in_mem(pm, dt)(addr)(s)

Repeatedly Skolemizing and flattening,
Expanding the definition of valid_in_mem,
Expanding the definition of read_data,
Expanding the definition of ok_lift,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using valid_iff_from_byte, matching in * where idt gets dt!1,
This completes the proof of `read_data_valid_in_mem`.
Q.E.D.

C.2 Proofs for Abstract_Read_Write_Plain (plain_memory.pvs)

This theory contains no provable formal statements.

C.3 Proofs for Address_Datatype (paging-data.pvs)

C.3.1 Address_Datatype.address_data_type_TCC1

Terse proof for `address_data_type_TCC1`.
`address_data_type_TCC1`:

{1} ∃ (x_1: (pod_data_type?[Memory_Address_4G])): TRUE

Using lemma address_data_type_exists,
This completes the proof of `address_data_type_TCC1`.
Q.E.D.

C.4 Proofs for Address_Helpers (constants.pvs)

C.4.1 Address_Helpers.type_add_rewrite

Terse proof for `type_add_rewrite`.
`type_add_rewrite`:

{1} ∀ (m_addr: Memory_Address, size: int): (m_addr + size)`type_of = m_addr`type_of

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `type_add_rewrite`.
Q.E.D.

C.4.2 Address_Helpers.offset_add_rewrite

Terse proof for `offset_add_rewrite`.
`offset_add_rewrite:`

{1} ∀ (m_addr: Memory_Address, size: int):
(m_addr + size)‘offset = m_addr‘offset + size

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `offset_add_rewrite`.
Q.E.D.

C.4.3 Address_Helpers.type_add_memory

Terse proof for `type_add_memory`.
`type_add_memory:`

{1} ∀ (m_addr, size): Mem?((m_addr + size)‘type_of)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `type_add_memory`.
Q.E.D.

C.4.4 Address_Helpers.address_add_zero

Terse proof for `address_add_zero`.
`address_add_zero:`

{1} ∀ (a: Address): a + 0 = a

Expanding the definition of +,
which is trivially true.
This completes the proof of `address_add_zero`.
Q.E.D.

C.4.5 Address_Helpers.address_add_sum

Terse proof for `address_add_sum`.
`address_add_sum:`

{1} ∀ (a: Address, i, j: int): (a + i) + j = a + (i + j)

Expanding the definition of +,
which is trivially true.
This completes the proof of `address_add_sum`.
Q.E.D.

C.4.6 Address_Helpers.reg_size_TCC1

Terse proof for `reg_size_TCC1`.

`reg_size_TCC1:`

{1} ∀ (max: Memory_Address, id: Register_Id):
¬ SYSENTER_EIP?(id) ∧
¬ SYSENTER_ESP?(id) ∧
¬ SYSENTER_CS?(id) ∧
¬ TR?(id) ∧
¬ LDTR?(id) ∧
¬ GDTR?(id) ∧ ¬ IDTR?(id) ∧ ¬ EFLAGS?(id) ∧ ¬ IP?(id) ∧ ¬ Mem?(id)
⊃ ¬ (GP?(id) ∧ Segment_Reg?(id)) ∧ ¬ (Segment_Reg?(id) ∧ Control_Reg?(id))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `reg_size_TCC1`.
Q.E.D.

C.4.7 Address_Helpers.reg_size_TCC2

Terse proof for `reg_size_TCC2`.

`reg_size_TCC2:`

{1} ∀ (max: Memory_Address, id: Register_Id):
¬ SYSENTER_EIP?(id) ∧
¬ SYSENTER_ESP?(id) ∧
¬ SYSENTER_CS?(id) ∧
¬ TR?(id) ∧
¬ LDTR?(id) ∧
¬ GDTR?(id) ∧ ¬ IDTR?(id) ∧ ¬ EFLAGS?(id) ∧ ¬ IP?(id) ∧ ¬ Mem?(id)
⊃ GP?(id) ∨ Segment_Reg?(id) ∨ Control_Reg?(id)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `reg_size_TCC2`.
Q.E.D.

C.5 Proofs for Address_Model (paging-data-models.pvs)

C.5.1 Address_Model.address_to_byte_TCC1

Terse proof for `address_to_byte_TCC1`.

`address_to_byte_TCC1:`

{1} ∀ (addr: Memory_Address_4G): offset(addr) ≥ 0

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less max_linear_offset bus_width
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `address_to_byte_TCC1`.
Q.E.D.

C.5.2 Address_Model.address_to_byte_TCC2

Terse proof for `address_to_byte_TCC2`.
`address_to_byte_TCC2`:

```
{1}  ∀ (addr: Memory_Address_4G): cut_bits(offset(addr), 0, 8) < max_byte
```

Repeatedly Skolemizing and flattening,
Using lemma cut_bits_below,
Rewriting using less_than_max_byte, matching in *,
This completes the proof of `address_to_byte_TCC2`.
Q.E.D.

C.5.3 Address_Model.address_to_byte_TCC3

Terse proof for `address_to_byte_TCC3`.
`address_to_byte_TCC3`:

```
{1}  ∀ (addr: Memory_Address_4G):
       every[nat]
           ({s: nat | s < max_byte})
           ((:cut_bits(offset(addr), 8, 8), cut_bits(offset(addr), 16, 8),
                cut_bits(offset(addr), 24, 8):))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less max_linear_offset bus_width
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `address_to_byte_TCC3`.
Q.E.D.

C.5.4 Address_Model.address_from_byte_TCC1

Terse proof for `address_from_byte_TCC1`.
`address_from_byte_TCC1`:

```
{1}  ∀ (data: list[Byte], a: Address): address_valid?(data, a) ⊃ 0 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `address_from_byte_TCC1`.
Q.E.D.

C.5.5 Address_Model.address_from_byte_TCC2

Terse proof for `address_from_byte_TCC2`.
`address_from_byte_TCC2`:

```
{1}  ∀ (data: list[Byte], a: Address): address_valid?(data, a) ⊃ 1 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `address_from_byte_TCC2`.
Q.E.D.

C.5.6 Address_Model.address_from_byte_TCC3

Terse proof for `address_from_byte_TCC3`.

`address_from_byte_TCC3:`

```
|-------
{1}   ∀ (data: list[Byte], a: Address): address_valid?(data, a) ⊃ 2 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `address_from_byte_TCC3`.

Q.E.D.

C.5.7 Address_Model.address_from_byte_TCC4

Terse proof for `address_from_byte_TCC4`.

`address_from_byte_TCC4:`

```
|-------
{1}   ∀ (data: list[Byte], a: Address): address_valid?(data, a) ⊃ 3 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `address_from_byte_TCC4`.

Q.E.D.

C.5.8 Address_Model.address_from_byte_TCC5

Terse proof for `address_from_byte_TCC5`.

`address_from_byte_TCC5:`

{1} ∀ (data: list[Byte], a: Address):
address_valid?(data, a) ⊃
Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
(0,
nt
(d
0
8)
nth
(dat
8,
8),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))‘type_of)
∧
0 ≤
Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
(0,
nth[By
(data,
0,
8),
nth[Byte
(data, 1
8,
8),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))‘offset
∧
Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
(0,
nth[By
(data,
0,
8),
nth[Byte
(data, 1
8,
8),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))‘offset
< max_linear_offset

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less address_valid? max_linear_offset bus_width

Expanding the definition of Mem,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `address_from_byte_TCC5`.

Q.E.D.

C.5.9 Address_Model.address_model_TCC1

Terse proof for `address_model_TCC1`.

`address_model_TCC1:`

```
{1}  ∀ (f: [[Memory_Address_4G, Address] → [list[Byte] → bool]]):
       (f = (λ (d: Memory_Address_4G, a: Address): (zero_mask?(4)))) ⊃
        (∀ (x1: [Memory_Address_4G, Address]): singleton?[list[Byte]](f(x1)))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Expanding the definition of singleton?,
which is trivially true.
This completes the proof of `address_model_TCC1`.
Q.E.D.

C.5.10 Address_Model.address_valid_to_byte

Terse proof for `address_valid_to_byte`.

`address_valid_to_byte:`

```
{1}  ∀ (d: Memory_Address_4G, a: Address):
       valid?(uidt(address_model))(to_byte(address_model)(d, a), a)
```

Installing automatic rewrites from: address_model address_to_byte length address_valid?
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of `address_valid_to_byte`.
Q.E.D.

C.5.11 Address_Model.address_is_pod

Terse proof for `address_is_pod`.

`address_is_pod:`

```
{1}  pod_data_type?(address_model)
```

Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Expanding the definition of uninterpreted_data_type?,
Applying propositional simplification,
we get 6 subgoals:

`address_is_pod.1:`

```
{1}  ∀ (l: list[Byte], a: Address):
       ¬ length(l) = size(address_model`uidt) ⊃ ¬ valid?(address_model`uidt)(l, a)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of address_model,

Expanding the definition of address_valid?,
which is trivially true.
This completes the proof of `address_is_pod.1`.
`address_is_pod.2`:

```
{1}  ∀ (d: Memory_Address_4G, a: Address):
       valid?(uidt(address_model))(to_byte(address_model)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Rewriting using address_valid_to_byte, matching in *,
This completes the proof of `address_is_pod.2`.
`address_is_pod.3`:

```
{1}  ∀ (l: list[Byte], a: Address):
       valid?(uidt(address_model))(l, a) ≡ up?(from_byte(address_model)(l, a))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of address_model,
Expanding the definition of address_from_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `address_is_pod.3`.
`address_is_pod.4`:

```
{1}  ∀ (d: Memory_Address_4G, a: Address):
       down(from_byte(address_model)(to_byte(address_model)(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: address_model address_from_byte address_to_byte address_valid? length nth overwrite_shift_bits_zero cut_bits_cut_bits overwrite_shift_bits_merge max_linear_offset bus_width zero_aligned
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using shift_bits_left_cut_bits, matching in *,
Applying extensionality,
we get 2 subgoals:
`address_is_pod.4.1`:

```
{-1}  Mem?(d'‘type_of)
{-2}  0 ≤ d'‘offset
{-3}  d'‘offset < expt(2, 32)
------------------------------------------
{1}   Mem(offset(d'))‘offset = d'‘offset
```

Expanding the definition of Mem,
which is trivially true.
This completes the proof of `address_is_pod.4.1`.
`address_is_pod.4.2`:

```
{-1}  Mem?(d'‘type_of)
{-2}  0 ≤ d'‘offset
{-3}  d'‘offset < expt(2, 32)
------------------------------------------
{1}   Mem(offset(d'))‘type_of = d'‘type_of
```

Expanding the definition of Mem,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `address_is_pod.4.2`.

address_is_pod.5:

{1} ∀ (l: list[Byte], a1, a2: Address):
valid?(uidt(address_model))(l, a1) ≡ valid?(uidt(address_model))(l, a2)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of address_model,
Expanding the definition of address_valid?,
which is trivially true.
This completes the proof of address_is_pod.5.
address_is_pod.6:

{1} size(uidt(address_model)) > 0

Expanding the definition of address_model,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of address_is_pod.6.
Q.E.D.

C.6 Proofs for Address_Util (memory.pvs)

C.6.1 Address_Util.address_block_subset_1

Terse proof for address_block_subset_1.
address_block_subset_1:

{1} ∀ (addr: Address, size: nat):
(address_block(addr + 1, size) ⊆ address_block(addr, 1 + size))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of address_block_subset_1.
Q.E.D.

C.6.2 Address_Util.address_block_subset_2

Terse proof for address_block_subset_2.
address_block_subset_2:

{1} ∀ (addresses: PRED[Address], addr: Address, size: nat):
(address_block(addr, size + 1) ⊆ addresses) ⊃
(address_block(addr + 1, size) ⊆ addresses)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of address_block_subset_2.
Q.E.D.

C.6.3 Address_Util.blocks_disjoint_disjoint

Terse proof for blocks_disjoint_disjoint.

`blocks_disjoint_disjoint:`

{1}	∀ (addr1, addr2: Address, size1, size2: nat): size1 > 0 ∧ size2 > 0 ⊃ blocks_disjoint?(addr1, size1, addr2, size2) = disjoint?(address_block(addr1, size1), address_block(addr2, size2))

Repeatedly Skolemizing and flattening,
Converting equality to IFF,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`blocks_disjoint_disjoint.1:`

{-1}	size1′ ≥ 0
{-2}	size2′ ≥ 0
{-3}	size1′ > 0
{-4}	size2′ > 0
{-5}	blocks_disjoint?(addr1′, size1′, addr2′, size2′)
{1}	disjoint?(address_block(addr1′, size1′), address_block(addr2′, size2′))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `blocks_disjoint_disjoint.1`.
`blocks_disjoint_disjoint.2:`

{-1}	size1′ ≥ 0
{-2}	size2′ ≥ 0
{-3}	size1′ > 0
{-4}	size2′ > 0
{-5}	disjoint?(address_block(addr1′, size1′), address_block(addr2′, size2′))
{1}	blocks_disjoint?(addr1′, size1′, addr2′, size2′)

Expanding the definition of blocks_disjoint?,
Applying disjunctive simplification to flatten sequent,
Expanding the definition(s) of (disjoint? empty? intersection member),
Case splitting on addr1!1 < addr2!1,
we get 2 subgoals:
`blocks_disjoint_disjoint.2.1:`

{-1}	addr1′ < addr2′
{-2}	size1′ ≥ 0
{-3}	size2′ ≥ 0
{-4}	size1′ > 0
{-5}	size2′ > 0
{-6}	type_of(addr1′) = type_of(addr2′)
{-7}	∀ (x: Address): ¬ (address_block(addr1′, size1′)(x) ∧ address_block(addr2′, size2′)(x))
{1}	addr1′ + size1′ ≤ addr2′
{2}	addr2′ + size2′ ≤ addr1′

Instantiating the top quantifier in -7 with the terms: addr2′,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `blocks_disjoint_disjoint.2.1`.

`blocks_disjoint_disjoint.2.2:`

{-1} $\text{size1}' \geq 0$
{-2} $\text{size2}' \geq 0$
{-3} $\text{size1}' > 0$
{-4} $\text{size2}' > 0$
{-5} $\text{type\_of}(\text{addr1}') = \text{type\_of}(\text{addr2}')$
{-6} $\forall\ (x\texttt{: Address}):$
$\neg\ (\text{address\_block}(\text{addr1}', \text{size1}')(x) \wedge \text{address\_block}(\text{addr2}', \text{size2}')(x))$

{1} $\text{addr1}' < \text{addr2}'$
{2} $\text{addr1}' + \text{size1}' \leq \text{addr2}'$
{3} $\text{addr2}' + \text{size2}' \leq \text{addr1}'$

Instantiating the top quantifier in -6 with the terms: addr1′,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `blocks_disjoint_disjoint.2.2`.
Q.E.D.

C.6.4 Address_Util.blocks_disjoint_symmetric

Terse proof for `blocks_disjoint_symmetric`.

`blocks_disjoint_symmetric:`

{1} $\forall$ (addr1, addr2: Address, size1, size2: nat):
blocks_disjoint?(addr1, size1, addr2, size2) = blocks_disjoint?(addr2, size2, addr1, size1)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `blocks_disjoint_symmetric`.
Q.E.D.

C.6.5 Address_Util.blocks_in_larger_set

Terse proof for `blocks_in_larger_set`.

`blocks_in_larger_set:`

{1} $\forall$ (addr1, addr2: Address, size1, size2: nat, addresses: PRED[Address]):
$(\text{address\_block}(\text{addr1}, \text{size1}) \subseteq \text{addresses}) \wedge$
$(\text{address\_block}(\text{addr2}, \text{size2}) \subseteq \text{addresses}) \wedge$
blocks_disjoint?(addr1, size1, addr2, size2)
$\supset\ (\text{address\_block}(\text{addr2}, \text{size2}) \subseteq (\text{addresses} \setminus \text{address\_block}(\text{addr1}, \text{size1})))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `blocks_in_larger_set`.
Q.E.D.

C.6.6 Address_Util.blocks_pairwise_disjoint_add

Terse proof for `blocks_pairwise_disjoint_add`.

`blocks_pairwise_disjoint_add:`

```
{1}  ∀ (addr: Address, size: posnat, blocks: list[[Address, posnat]]):
       blocks_pairwise_disjoint(blocks) ∧
        (∀ (oaddr: Address, osize: posnat):
             member((oaddr, osize), blocks) ⊃ blocks_disjoint?(addr, size, oaddr, osize))
        ⊃ blocks_pairwise_disjoint(cons((addr, size), blocks))
```

Expanding the definition of blocks_pairwise_disjoint,
Repeatedly Skolemizing and flattening,
Expanding the definition of member,
Case splitting on addr!2 = addr!1 AND size!2 = size!1,
we get 2 subgoals:
`blocks_pairwise_disjoint_add.1:`

```
{-1}  addr'' = addr' ∧ size'' = size'
{-2}  size'' > 0
{-3}  size''' > 0
{-4}  size' > 0
{-5}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-6}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
{-7}  addr'' = addr' ∧ size'' = size' ∨ member((addr'', size''), blocks')
{-8}  member((addr''', size'''),
                list_remove((addr'', size''), cons((addr', size'), blocks')))
------------------------------------------------------------------------
{1}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of list_remove,
Rewriting using member_list_remove, matching in *,
Instantiating the top quantifier in -7 with the terms: addr''', size''',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `blocks_pairwise_disjoint_add.1`.
`blocks_pairwise_disjoint_add.2:`

```
{-1}  size'' > 0
{-2}  size''' > 0
{-3}  size' > 0
{-4}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-5}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
{-6}  addr'' = addr' ∧ size'' = size' ∨ member((addr'', size''), blocks')
{-7}  member((addr''', size'''),
                list_remove((addr'', size''), cons((addr', size'), blocks')))
------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Splitting conjunctions,
we get 2 subgoals:

`blocks_pairwise_disjoint_add.2.1:`

```
{-1}  addr'' = addr' ∧ size'' = size'
{-2}  size'' > 0
{-3}  size''' > 0
{-4}  size' > 0
{-5}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-6}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
{-7}  member((addr''', size'''),
                list_remove((addr'', size''), cons((addr', size'), blocks')))
------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

which is trivially true.
This completes the proof of `blocks_pairwise_disjoint_add.2.1`.
`blocks_pairwise_disjoint_add.2.2:`

```
{-1}  member((addr'', size''), blocks')
{-2}  size'' > 0
{-3}  size''' > 0
{-4}  size' > 0
{-5}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-6}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
{-7}  member((addr''', size'''),
                list_remove((addr'', size''), cons((addr', size'), blocks')))
------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Expanding the definition of list_remove,
Lifting IF-conditions to the top level,
Splitting conjunctions,
we get 2 subgoals:
`blocks_pairwise_disjoint_add.2.2.1:`

```
{-1}  (addr'' = addr' ∧ size'' = size') ∧
       member((addr''', size'''), list_remove((addr'', size''), blocks'))
{-2}  member((addr'', size''), blocks')
{-3}  size'' > 0
{-4}  size''' > 0
{-5}  size' > 0
{-6}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-7}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `blocks_pairwise_disjoint_add.2.2.1`.

`blocks_pairwise_disjoint_add.2.2.2:`

```
{-1}  ¬ (addr'' = addr' ∧ size'' = size') ∧
       member((addr''', size'''),
              cons((addr', size'), list_remove((addr'', size''), blocks')))
{-2}  member((addr'', size''), blocks')
{-3}  size'' > 0
{-4}  size''' > 0
{-5}  size' > 0
{-6}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-7}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
-------------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Applying disjunctive simplification to flatten sequent,

Hiding formulas: 1,

Expanding the definition of member,

Splitting conjunctions,

we get 2 subgoals:

`blocks_pairwise_disjoint_add.2.2.2.1:`

```
{-1}  addr''' = addr' ∧ size''' = size'
{-2}  member((addr'', size''), blocks')
{-3}  size'' > 0
{-4}  size''' > 0
{-5}  size' > 0
{-6}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-7}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
-------------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -8 with the terms: addr'', size'',

Simplifying, rewriting, and recording with decision procedures,

Rewriting using blocks_disjoint_symmetric, matching in *,

This completes the proof of `blocks_pairwise_disjoint_add.2.2.2.1`.

`blocks_pairwise_disjoint_add.2.2.2.2:`

```
{-1}  member((addr''', size'''), list_remove((addr'', size''), blocks'))
{-2}  member((addr'', size''), blocks')
{-3}  size'' > 0
{-4}  size''' > 0
{-5}  size' > 0
{-6}  ∀ (addr_1, addr_2: Address, size_1, size_2: posnat):
        member((addr_1, size_1), blocks') ∧
         member((addr_2, size_2), list_remove((addr_1, size_1), blocks'))
         ⊃ blocks_disjoint?(addr_1, size_1, addr_2, size_2)
{-7}  ∀ (oaddr: Address, osize: posnat):
        member((oaddr, osize), blocks') ⊃ blocks_disjoint?(addr', size', oaddr, osize)
-------------------------------------------------------------------------------
{1}   addr'' = addr' ∧ size'' = size'
{2}   blocks_disjoint?(addr'', size'', addr''', size''')
```

Instantiating the top quantifier in -6 with the terms: addr'', addr''', size'', size''',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `blocks_pairwise_disjoint_add.2.2.2.2`.
Q.E.D.

C.6.7 Address_Util.blocks_pairwise_disjoint_remove

Terse proof for `blocks_pairwise_disjoint_remove`.

`blocks_pairwise_disjoint_remove:`

```
-------------------------------------------------------------------------------
{1}   ∀ (addr: Address, size: posnat, blocks: list[[Address, posnat]]):
        blocks_pairwise_disjoint(blocks) ⊃
         blocks_pairwise_disjoint(list_remove((addr, size), blocks))
```

Expanding the definition of blocks_pairwise_disjoint,
Repeatedly Skolemizing and flattening,
Rewriting using list_remove_commutative, matching in *,
Rewriting using member_list_remove, matching in *,
Rewriting using member_list_remove, matching in *,
Instantiating the top quantifier in -4 with the terms: addr'', addr''', size'', size''',
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `blocks_pairwise_disjoint_remove`.
Q.E.D.

C.6.8 Address_Util.subset_block_is_free

Terse proof for `subset_block_is_free`.

`subset_block_is_free:`

```
-------------------------------------------------------------------------------
{1}   ∀ (b_1, b_2: PRED[Address], alloc_points: list[[Address, posnat]]):
        (b_1 ⊆ b_2) ∧ block_is_free(b_2, alloc_points) ⊃ block_is_free(b_1, alloc_points)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of block_is_free,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,

Using lemma disjoint_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_equal, matching in *,
This completes the proof of `subset_block_is_free`.
Q.E.D.

C.6.9 Address_Util.block_is_free_add

Terse proof for `block_is_free_add`.
`block_is_free_add`:

{1} ∀ (block: PRED⌊Address⌋, alloc_points: list⌊⌊Address, posnat⌋⌋, addr: Address, size: posnat): block_is_free(block, alloc_points) ∧ disjoint?(block, address_block(addr, size)) ⊃ block_is_free(block, cons((addr, size), alloc_points))

Repeatedly Skolemizing and flattening,
Expanding the definition of block_is_free,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: member
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: addr″, size″,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `block_is_free_add`.
Q.E.D.

C.7 Proofs for Alignment (vfiasco-prelude.pvs)

C.7.1 Alignment.aligned_zero

Terse proof for `aligned_zero`.
`aligned_zero`:

{1} ∀ (n: nat): aligned?(n)(0)

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Rewriting using divides_zero, matching in *,
This completes the proof of `aligned_zero`.
Q.E.D.

C.7.2 Alignment.zero_aligned

Terse proof for `zero_aligned`.
`zero_aligned`:

{1} ∀ (n: nat): aligned?(0)(n)

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Expanding the definition of expt,

Rewriting using one_divides, matching in *,
This completes the proof of `zero_aligned`.
Q.E.D.

C.7.3 Alignment.aligned_bigger

Terse proof for `aligned_bigger`.
`aligned_bigger:`

$$\frac{}{\{1\}\quad \forall\ (m,\ n_1,\ n_2\colon \text{nat})\colon\ n_2 \le n_1 \wedge \text{aligned?}(n_1)(m) \supset \text{aligned?}(n_2)(m)}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Forward chaining on divides_expt_gt,
This completes the proof of `aligned_bigger`.
Q.E.D.

C.7.4 Alignment.aligned_plus

Terse proof for `aligned_plus`.
`aligned_plus:`

$$\frac{}{\{1\}\quad \forall\ (m,\ n\colon \text{nat},\ a\colon \text{aligned}(n),\ b\colon \text{aligned}(m))\colon\ \text{aligned?}(\min(n,\ m))(a+b)}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of min,
Expanding the definition of aligned?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`aligned_plus.1:`

$$\frac{\begin{array}{ll}\{-1\} & m' \ge 0\\ \{-2\} & n' \ge 0\\ \{-3\} & \text{divides}(\text{expt}(2,\ n'),\ a')\\ \{-4\} & \text{divides}(\text{expt}(2,\ m'),\ b')\\ \{-5\} & n' > m'\end{array}}{\{1\}\quad \text{divides}(\text{expt}(2,\ m'),\ a'+b')}$$

Rewriting using divides_sum, matching in *,
Hiding formulas: -4, 2,
Using lemma expt_divides,
Simplifying, rewriting, and recording with decision procedures,
Using lemma divides_transitive,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_plus.1`.

`aligned_plus.2`:

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	divides(expt(2, n'), a')
{-4}	divides(expt(2, m'), b')
{1}	$n' > m'$
{2}	divides(expt(2, n'), $a' + b'$)

Rewriting using divides_sum, matching in *,
Hiding formulas: -3, 3,
Using lemma expt_divides,
Simplifying, rewriting, and recording with decision procedures,
Using lemma divides_transitive,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_plus.2`.
Q.E.D.

C.7.5 Alignment.aligned_plus_n

Terse proof for `aligned_plus_n`.
`aligned_plus_n`:

{1}	$\forall$ (n: nat, a, b: aligned(n)): aligned?(n)($a + b$)

Repeatedly Skolemizing and flattening,
Using lemma aligned_plus,
Expanding the definition of min,
which is trivially true.
This completes the proof of `aligned_plus_n`.
Q.E.D.

C.7.6 Alignment.aligned_minus_TCC1

Terse proof for `aligned_minus_TCC1`.
`aligned_minus_TCC1`:

{1}	$\forall$ (m, n: nat, a: aligned(n), b: aligned(m)): $a \geq b \supset a - b \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `aligned_minus_TCC1`.
Q.E.D.

C.7.7 Alignment.aligned_minus

Terse proof for `aligned_minus`.
`aligned_minus`:

{1}	$\forall$ (m, n: nat, a: aligned(n), b: aligned(m)): $a \geq b \supset$ aligned?(min(n, m))($a - b$)

Repeatedly Skolemizing and flattening,

Expanding the definition of min,
Expanding the definition of aligned?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`aligned_minus.1:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	divides(expt(2, n'), a')
{-4}	divides(expt(2, m'), b')
{-5}	$a' \geq b'$
{-6}	$n' > m'$
{1}	divides(expt(2, m'), $a' - b'$)

Rewriting using divides_diff, matching in *,
Hiding formulas: -4, 2,
Using lemma expt_divides,
Simplifying, rewriting, and recording with decision procedures,
Using lemma divides_transitive,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_minus.1`.
`aligned_minus.2:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	divides(expt(2, n'), a')
{-4}	divides(expt(2, m'), b')
{-5}	$a' \geq b'$
{1}	$n' > m'$
{2}	divides(expt(2, n'), $a' - b'$)

Rewriting using divides_diff, matching in *,
Hiding formulas: -3, 3,
Using lemma expt_divides,
Simplifying, rewriting, and recording with decision procedures,
Using lemma divides_transitive,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_minus.2`.
Q.E.D.

C.7.8 Alignment.aligned_mult_expt_2

Terse proof for `aligned_mult_expt_2`.
`aligned_mult_expt_2:`

{1}	$\forall$ (m, n: nat, a: aligned(n)): aligned?($n + m$)($a \times$ expt(2, m))

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Rewriting using expt_plus_aux, matching in *,
Using lemma divides_prod_elim1,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_mult_expt_2`.
Q.E.D.

C.7.9 Alignment.aligned_rem

Terse proof for `aligned_rem`.

`aligned_rem:`

{1}	$\forall$ (m, n, i: nat): aligned?(n)(m) $\supset$ aligned?(n)(rem(expt(2, i))(m))

Repeatedly Skolemizing and flattening,

Expanding the definition of aligned?,

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: expt(2, n'), rem(expt(2, i'))(m'), 0,

we get 2 subgoals:

`aligned_rem.1:`

{-1}	rem(expt(2, n'))(rem(expt(2, i'))(m')) = 0 $\equiv$ divides(expt(2, n'), rem(expt(2, i'))(m') − 0)
{-2}	$m' \geq 0$
{-3}	$n' \geq 0$
{-4}	$i' \geq 0$
{-5}	divides(expt(2, n'), m')
{1}	divides(expt(2, n'), rem(expt(2, i'))(m'))

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: 2,

Case splitting on i!1 <= n!1,

we get 2 subgoals:

`aligned_rem.1.1:`

{-1}	$i' \leq n'$
{-2}	$m' \geq 0$
{-3}	$n' \geq 0$
{-4}	$i' \geq 0$
{-5}	divides(expt(2, n'), m')
{1}	rem(expt(2, n'))(rem(expt(2, i'))(m')) = 0

Rewriting using rem_mod, matching in *,

we get 2 subgoals:

`aligned_rem.1.1.1:`

{-1}	$i' \leq n'$
{-2}	$m' \geq 0$
{-3}	$n' \geq 0$
{-4}	$i' \geq 0$
{-5}	divides(expt(2, n'), m')
{1}	rem(expt(2, i'))(m') = 0

Using lemma rem_def2,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on divides_expt_gt,

This completes the proof of `aligned_rem.1.1.1`.

`aligned_rem.1.1.2:`

{-1}	$i' \leq n'$
{-2}	$m' \geq 0$
{-3}	$n' \geq 0$
{-4}	$i' \geq 0$
{-5}	divides(expt(2, n'), m')
{1}	rem(expt(2, i'))(m') $<$ expt(2, n')
{2}	rem(expt(2, n'))(rem(expt(2, i'))(m')) $=$ 0

Using lemma both_sides_expt_gt1_le,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_rem.1.1.2`.
`aligned_rem.1.2:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	divides(expt(2, n'), m')
{1}	$i' \leq n'$
{2}	rem(expt(2, n'))(rem(expt(2, i'))(m')) $=$ 0

Rewriting using rem_rem, matching in *,
we get 2 subgoals:
`aligned_rem.1.2.1:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	divides(expt(2, n'), m')
{1}	$i' \leq n'$
{2}	rem(expt(2, n'))(m') $=$ 0

Rewriting using rem_def2, matching in *,
This completes the proof of `aligned_rem.1.2.1`.
`aligned_rem.1.2.2:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	divides(expt(2, n'), m')
{1}	divides(expt(2, n'), expt(2, i'))
{2}	$i' \leq n'$
{3}	rem(expt(2, n'))(rem(expt(2, i'))(m')) $=$ 0

Rewriting using expt_divides, matching in *,
This completes the proof of `aligned_rem.1.2.2`.
`aligned_rem.2:`

{-1}	$m' \geq 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	divides(expt(2, n'), m')
{1}	$0 <$ expt(2, n')
{2}	divides(expt(2, n'), rem(expt(2, i'))(m'))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_rem.2`.

Q.E.D.

C.7.10 Alignment.aligned_add_below

Terse proof for `aligned_add_below`.

`aligned_add_below:`

{1}	$\forall\ (n,\ m,\ i,\ k\colon \text{nat}):$ $i \le k \wedge n < \text{expt}(2,\ k) \wedge \text{aligned?}(i)(n) \wedge m < \text{expt}(2,\ i) \supset$ $n + m < \text{expt}(2,\ k)$

Repeatedly Skolemizing and flattening,

Expanding the definition of aligned?,

Expanding the definition of divides,

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Applying bit_split_less

Instantiating the top quantifier in -1 with the terms: x', m', $k' - i'$, i',

we get 3 subgoals:

`aligned_add_below.1:`

{-1}	$m' < \text{expt}(2,\ i') \supset$ $(x' \times \text{expt}(2,\ i') + m' < \text{expt}(2,\ i' + k' - i') \equiv$ $x' < \text{expt}(2,\ k' - i'))$
{-2}	$\text{integer\_pred}(x')$
{-3}	$\text{expt}(2,\ i') \times x' \ge 0$
{-4}	$m' \ge 0$
{-5}	$i' \ge 0$
{-6}	$k' \ge 0$
{-7}	$i' \le k'$
{-8}	$\text{expt}(2,\ i') \times x' < \text{expt}(2,\ k')$
{-9}	$n' = \text{expt}(2,\ i') \times x'$
{-10}	$m' < \text{expt}(2,\ i')$
{1}	$\text{expt}(2,\ i') \times x' + m' < \text{expt}(2,\ k')$

Simplifying, rewriting, and recording with decision procedures,

Applying both_sides_times_pos_lt2

Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, i')$, x', $\text{expt}(2, k' - i')$,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using expt_plus, matching in *,

This completes the proof of `aligned_add_below.1`.

`aligned_add_below.2:`

{-1}	integer_pred(x')
{-2}	expt(2, i') $\times$ x' $\geq$ 0
{-3}	m' $\geq$ 0
{-4}	i' $\geq$ 0
{-5}	k' $\geq$ 0
{-6}	i' $\leq$ k'
{-7}	expt(2, i') $\times$ x' $<$ expt(2, k')
{-8}	n' = expt(2, i') $\times$ x'
{-9}	m' $<$ expt(2, i')
{1}	$k' - i'$ $\geq$ 0
{2}	expt(2, i') $\times$ $x' + m'$ $<$ expt(2, k')

Applying both_sides_expt_gt1_le
Instantiating the top quantifier in -1 with the terms: 2, i', k',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_add_below.2`.
`aligned_add_below.3:`

{-1}	integer_pred(x')
{-2}	expt(2, i') $\times$ x' $\geq$ 0
{-3}	m' $\geq$ 0
{-4}	i' $\geq$ 0
{-5}	k' $\geq$ 0
{-6}	i' $\leq$ k'
{-7}	expt(2, i') $\times$ x' $<$ expt(2, k')
{-8}	n' = expt(2, i') $\times$ x'
{-9}	m' $<$ expt(2, i')
{1}	x' $\geq$ 0
{2}	expt(2, i') $\times$ $x' + m'$ $<$ expt(2, k')

Applying pos_times_le
Instantiating the top quantifier in -1 with the terms: expt(2, i'), x',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_add_below.3`.
Q.E.D.

C.8 Proofs for Allocation_Info (allocators.pvs)

C.8.1 Allocation_Info.consistent_allocator_state_change

Terse proof for `consistent_allocator_state_change`.
`consistent_allocator_state_change:`

{1}	$\forall$ (ai: Allocator_Info, pm: (plain_memory?), ac: (allocator?(pm)), s_1, s_2: State): consistent_allocator?(pm)(ac, ai)(s_1) $\wedge$ ac`allocated(pm)($s_1$) = ac`allocated(pm)(s_2) $\supset$ consistent_allocator?(pm)(ac, ai)(s_2)

Repeatedly Skolemizing and flattening,
Expanding the definition of consistent_allocator?,
Applying disjunctive simplification to flatten sequent,

Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `consistent_allocator_state_change`.
Q.E.D.

C.8.2 Allocation_Info.consistent_allocator_read_TCC1

Terse proof for `consistent_allocator_read_TCC1`.

`consistent_allocator_read_TCC1:`

```
{1}  ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)),
          addr: Address):
       pm`states(s) ∧
        consistent_allocator?(pm)(ac, ai)(s) ∧ union(pm`ro_addr, pm`rw_addr)(addr)
        ⊃
        OK?[State, Byte](memory_read(pm`mem)(addr)(s)) ∨
         Exception?[State, Byte](memory_read(pm`mem)(addr)(s))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of `consistent_allocator_read_TCC1`.
Q.E.D.

C.8.3 Allocation_Info.consistent_allocator_read

Terse proof for `consistent_allocator_read`.

`consistent_allocator_read:`

```
{1}  ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?(pm)),
          addr: Address):
       pm`states(s) ∧
        consistent_allocator?(pm)(ac, ai)(s) ∧ union(pm`ro_addr, pm`rw_addr)(addr)
        ⊃ consistent_allocator?(pm)(ac, ai)(state(memory_read(pm`mem)(addr)(s)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of consistent_allocator?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_allocated_memory_read, matching in *,
This completes the proof of `consistent_allocator_read`.
Q.E.D.

C.8.4 Allocation_Info.consistent_allocator_write_TCC1

Terse proof for `consistent_allocator_write_TCC1`.

consistent_allocator_write_TCC1:

{1} ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)),
addr: Address, byte: Byte):
pm‘states(s) ∧
consistent_allocator?(pm)(ac, ai)(s) ∧
pm‘rw_addr(addr) ∧ ¬ ac‘private_mem(pm)(addr)
⊃
OK?[State, Unit](memory_write(pm‘mem)(addr, byte)(s)) ∨
Exception?[State, Unit](memory_write(pm‘mem)(addr, byte)(s))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_write_ok, matching in *,
This completes the proof of consistent_allocator_write_TCC1.
Q.E.D.

C.8.5 Allocation_Info.consistent_allocator_write

Terse proof for consistent_allocator_write.

consistent_allocator_write:

{1} ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?(pm)), addr: Address,
byte: Byte):
pm‘states(s) ∧
consistent_allocator?(pm)(ac, ai)(s) ∧
pm‘rw_addr(addr) ∧ ¬ ac‘private_mem(pm)(addr)
⊃ consistent_allocator?(pm)(ac, ai)(state(memory_write(pm‘mem)(addr, byte)(s)))

Repeatedly Skolemizing and flattening,
Expanding the definition of consistent_allocator?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_allocated_memory_write, matching in *,
This completes the proof of consistent_allocator_write.
Q.E.D.

C.8.6 Allocation_Info.consistent_allocator_alloc_TCC1

Terse proof for consistent_allocator_alloc_TCC1.

consistent_allocator_alloc_TCC1:

{1} ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)),
size: posnat):
pm‘states(s) ∧
consistent_allocator?(pm)(ac, ai)(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
OK?[State, Address](ac‘alloc(pm)(size)(s)) ∨
Exception?[State, Address](ac‘alloc(pm)(size)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `consistent_allocator_alloc_TCC1`.
Q.E.D.

C.8.7 Allocation_Info.consistent_allocator_alloc

Terse proof for `consistent_allocator_alloc`.
`consistent_allocator_alloc:`

```
{1}  ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?(pm)),
          size: posnat):
       pm‘states(s) ∧
        consistent_allocator?(pm)(ac, ai)(s) ∧
         OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
        ⊃
        consistent_allocator?(pm)(ac, cons((data(ac‘alloc(pm)(size)(s)), size), ai))
                                      (state(ac‘alloc(pm)(size)(s)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of consistent_allocator?,
Applying propositional simplification,
we get 2 subgoals:
`consistent_allocator_alloc.1:`

```
{-1}  plain_memory?(pm′)
{-2}  allocator?(pm′)(ac′)
{-3}  size′ > 0
{-4}  pm′‘states(s′)
{-5}  allocator_info?(ai′)
{-6}  ∀ (addr: Address, size: posnat):
        member((addr, size), ai′) ⊃
         (address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′))
{-7}  OK?(ac′‘alloc(pm′)(size′)(s′))
-------------------------------------------------------------------------------
{1}   allocator_info?(cons((data(ac′‘alloc(pm′)(size′)(s′)), size′), ai′))
{2}   data(ac′‘alloc(pm′)(size′)(s′)) = null_address
```

Expanding the definition of allocator_info?,
Rewriting using blocks_pairwise_disjoint_add, matching in *,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using blocks_disjoint_disjoint, matching in *,
Instantiating the top quantifier in -8 with the terms: oaddr′, osize′,
Simplifying, rewriting, and recording with decision procedures,
Using lemma disjoint_mono,
Installing automatic rewrites from: subset_equal
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma allocator_alloc_disjoint,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using disjoint_symmetric, matching in *,
This completes the proof of `consistent_allocator_alloc.1`.

`consistent_allocator_alloc.2:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac′)
{-3} size′ > 0
{-4} pm′‘states(s′)
{-5} allocator_info?(ai′)
{-6} ∀ (addr: Address, size: posnat):
member((addr, size), ai′) ⊃
(address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′))
{-7} OK?(ac′‘alloc(pm′)(size′)(s′))

{1} ∀ (addr: Address, size: posnat):
member((addr, size), cons((data(ac′‘alloc(pm′)(size′)(s′)), size′), ai′))
⊃
(address_block(addr, size) ⊆ ac′‘allocated(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
{2} data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Hiding formulas: -5,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Applying propositional simplification,

we get 2 subgoals:

`consistent_allocator_alloc.2.1:`

{-1} addr′ = data(ac′‘alloc(pm′)(size′)(s′))
{-2} size″ = size′
{-3} size″ > 0
{-4} plain_memory?(pm′)
{-5} allocator?(pm′)(ac′)
{-6} size′ > 0
{-7} pm′‘states(s′)
{-8} ∀ (addr: Address, size: posnat):
member((addr, size), ai′) ⊃
(address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′))
{-9} OK?(ac′‘alloc(pm′)(size′)(s′))

{1} (address_block(addr′, size″) ⊆ ac′‘allocated(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
{2} data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Rewriting using allocator_allocated_alloc, matching in *,

Replacing using formula -1,

Replacing using formula -2,

Rewriting using union_subset3, matching in *,

This completes the proof of `consistent_allocator_alloc.2.1`.

`consistent_allocator_alloc.2.2:`

{-1} member((addr′, size″), ai′)
{-2} size″ > 0
{-3} plain_memory?(pm′)
{-4} allocator?(pm′)(ac′)
{-5} size′ > 0
{-6} pm′‘states(s′)
{-7} ∀ (addr: Address, size: posnat):
member((addr, size), ai′) ⊃
(address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′))
{-8} OK?(ac′‘alloc(pm′)(size′)(s′))

{1} (address_block(addr′, size″) ⊆ ac′‘allocated(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
{2} data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using allocator_allocated_alloc, matching in *,

Rewriting using subset_bigger_union_left, matching in *,

This completes the proof of `consistent_allocator_alloc.2.2`.

Q.E.D.

C.8.8 Allocation_Info.consistent_allocator_free_TCC1

Terse proof for `consistent_allocator_free_TCC1`.

`consistent_allocator_free_TCC1:`

{1} ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)),
addr: Address):
OK?(ac‘free(pm)(addr)(s)) ∧
consistent_allocator?(pm)(ac, ai)(s) ∧ pm‘states(s)
⊃ up?[posnat](ac‘freed_size(pm)(addr)(s))

Repeatedly Skolemizing and flattening,

Rewriting using allocator_free_freed_size, matching in *,

This completes the proof of `consistent_allocator_free_TCC1`.

Q.E.D.

C.8.9 Allocation_Info.consistent_allocator_free_TCC2

Terse proof for `consistent_allocator_free_TCC2`.

`consistent_allocator_free_TCC2:`

```
{1}  ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?[State](pm)),
         addr: Address):
       pm‘states(s) ∧
        consistent_allocator?(pm)(ac, ai)(s) ∧
         OK?(ac‘free(pm)(addr)(s)) ∧
          (member((addr, down(ac‘freed_size(pm)(addr)(s))), ai) ∨
             blocks_pairwise_disjoint(cons((addr, down(ac‘freed_size(pm)(addr)(s))),
                                           ai)))
        ⊃
        OK?[State, Unit](ac‘free(pm)(addr)(s)) ∨
         Exception?[State, Unit](ac‘free(pm)(addr)(s))
```

Repeatedly Skolemizing and flattening,

This completes the proof of `consistent_allocator_free_TCC2`.

Q.E.D.

C.8.10 Allocation_Info.consistent_allocator_free

Terse proof for `consistent_allocator_free`.

`consistent_allocator_free:`

```
{1}  ∀ (ai: Allocator_Info, pm: (plain_memory?), s: State, ac: (allocator?(pm)),
         addr: Address):
       pm‘states(s) ∧
        consistent_allocator?(pm)(ac, ai)(s) ∧
         OK?(ac‘free(pm)(addr)(s)) ∧
          (member((addr, down(ac‘freed_size(pm)(addr)(s))), ai) ∨
             blocks_pairwise_disjoint(cons((addr, down(ac‘freed_size(pm)(addr)(s))),
                                           ai)))
        ⊃
        consistent_allocator?(pm)
                             (ac,
                               list_remove((addr, down(ac‘freed_size(pm)(addr)(s))), ai))
                             (state(ac‘free(pm)(addr)(s)))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of consistent_allocator?,

Splitting conjunctions,

we get 2 subgoals:

`consistent_allocator_free.1:`

```
{-1}  plain_memory?(pm′)
{-2}  allocator?(pm′)(ac′)
{-3}  pm′‘states(s′)
{-4}  allocator_info?(ai′) ∧
        (∀ (addr: Address, size: posnat):
             member((addr, size), ai′) ⊃
               (address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′)))
{-5}  OK?(ac′‘free(pm′)(addr′)(s′))
{-6}  (member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′) ∨
         blocks_pairwise_disjoint(cons((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))),
                                       ai′)))
------------------------------------------------------------------------------
{1}   allocator_info?(list_remove((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′))
```

Expanding the definition of allocator_info?,

Rewriting using blocks_pairwise_disjoint_remove, matching in *,

This completes the proof of `consistent_allocator_free.1`.

`consistent_allocator_free.2:`

```
{-1}  plain_memory?(pm′)
{-2}  allocator?(pm′)(ac′)
{-3}  pm′‘states(s′)
{-4}  allocator_info?(ai′) ∧
        (∀ (addr: Address, size: posnat):
             member((addr, size), ai′) ⊃
               (address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′)))
{-5}  OK?(ac′‘free(pm′)(addr′)(s′))
{-6}  (member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′) ∨
         blocks_pairwise_disjoint(cons((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))),
                                       ai′)))
------------------------------------------------------------------------------
{1}   ∀ (addr: Address, size: posnat):
        member((addr, size),
                  list_remove((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′))
        ⊃
        (address_block(addr, size) ⊆ ac′‘allocated(pm′)(state(ac′‘free(pm′)(addr′)(s′))))
```

Repeatedly Skolemizing and flattening,

Rewriting using member_list_remove, matching in *,

Splitting conjunctions,

we get 2 subgoals:

`consistent_allocator_free.2.1:`

{-1} size′ > 0
{-2} plain_memory?(pm′)
{-3} allocator?(pm′)(ac′)
{-4} pm′‘states(s′)
{-5} allocator_info?(ai′)
{-6} ∀ (addr: Address, size: posnat):
member((addr, size), ai′) ⊃
(address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′))
{-7} OK?(ac′‘free(pm′)(addr′)(s′))
{-8} (member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′) ∨
blocks_pairwise_disjoint(cons((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))),
ai′)))

{1} TRUE
{2} (address_block(addr″, size′) ⊆ ac′‘allocated(pm′)(state(ac′‘free(pm′)(addr′)(s′))))

which is trivially true.

This completes the proof of `consistent_allocator_free.2.1`.

`consistent_allocator_free.2.2:`

{-1} ¬ (addr″ = addr′ ∧ size′ = down(ac′‘freed_size(pm′)(addr′)(s′))) ∧
member((addr″, size′), ai′)
{-2} size′ > 0
{-3} plain_memory?(pm′)
{-4} allocator?(pm′)(ac′)
{-5} pm′‘states(s′)
{-6} allocator_info?(ai′)
{-7} ∀ (addr: Address, size: posnat):
member((addr, size), ai′) ⊃
(address_block(addr, size) ⊆ ac′‘allocated(pm′)(s′))
{-8} OK?(ac′‘free(pm′)(addr′)(s′))
{-9} (member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′) ∨
blocks_pairwise_disjoint(cons((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))),
ai′)))

{1} (address_block(addr″, size′) ⊆ ac′‘allocated(pm′)(state(ac′‘free(pm′)(addr′)(s′))))

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using allocator_allocated_free, matching in *,

Rewriting using disjoint_subset_difference, matching in *,

Hiding formulas: 3,

Splitting conjunctions,

we get 2 subgoals:

`consistent_allocator_free.2.2.1:`

{-1}	member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′)
{-2}	member((addr″, size′), ai′)
{-3}	size′ > 0
{-4}	plain_memory?(pm′)
{-5}	allocator?(pm′)(ac′)
{-6}	pm′‘states(s′)
{-7}	allocator_info?(ai′)
{-8}	(address_block(addr″, size′) ⊆ ac′‘allocated(pm′)(s′))
{-9}	OK?(ac′‘free(pm′)(addr′)(s′))
{1}	disjoint?(address_block(addr″, size′), address_block(addr′, down(ac′‘freed_size(pm′)(addr′)(s′))))
{2}	addr″ = addr′ ∧ size′ = down(ac′‘freed_size(pm′)(addr′)(s′))

Expanding the definition of allocator_info?,
Expanding the definition of blocks_pairwise_disjoint,
Instantiating the top quantifier in -7 with the terms: addr″, addr′, size′, down(ac′‘freed_size(pm′)(addr′)(s′)),
Simplifying, rewriting, and recording with decision procedures,
Splitting conjunctions,
we get 2 subgoals:
`consistent_allocator_free.2.2.1.1:`

{-1}	blocks_disjoint?(addr″, size′, addr′, down(ac′‘freed_size(pm′)(addr′)(s′)))
{-2}	member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′)
{-3}	member((addr″, size′), ai′)
{-4}	size′ > 0
{-5}	plain_memory?(pm′)
{-6}	allocator?(pm′)(ac′)
{-7}	pm′‘states(s′)
{-8}	(address_block(addr″, size′) ⊆ ac′‘allocated(pm′)(s′))
{-9}	OK?(ac′‘free(pm′)(addr′)(s′))
{1}	disjoint?(address_block(addr″, size′), address_block(addr′, down(ac′‘freed_size(pm′)(addr′)(s′))))
{2}	addr″ = addr′ ∧ size′ = down(ac′‘freed_size(pm′)(addr′)(s′))

Rewriting using blocks_disjoint_disjoint, matching in *,
This completes the proof of `consistent_allocator_free.2.2.1.1`.
`consistent_allocator_free.2.2.1.2:`

{-1}	member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), ai′)
{-2}	member((addr″, size′), ai′)
{-3}	size′ > 0
{-4}	plain_memory?(pm′)
{-5}	allocator?(pm′)(ac′)
{-6}	pm′‘states(s′)
{-7}	(address_block(addr″, size′) ⊆ ac′‘allocated(pm′)(s′))
{-8}	OK?(ac′‘free(pm′)(addr′)(s′))
{1}	member((addr′, down(ac′‘freed_size(pm′)(addr′)(s′))), list_remove((addr″, size′), ai′))
{2}	disjoint?(address_block(addr″, size′), address_block(addr′, down(ac′‘freed_size(pm′)(addr′)(s′))))
{3}	addr″ = addr′ ∧ size′ = down(ac′‘freed_size(pm′)(addr′)(s′))

Rewriting using member_list_remove, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `consistent_allocator_free.2.2.1.2`.

`consistent_allocator_free.2.2.2:`

```
{-1}  blocks_pairwise_disjoint(cons((addr', down(ac'`freed_size(pm')(addr')(s'))),
                                    ai'))
{-2}  member((addr'', size'), ai')
{-3}  size' > 0
{-4}  plain_memory?(pm')
{-5}  allocator?(pm')(ac')
{-6}  pm'`states(s')
{-7}  allocator_info?(ai')
{-8}  (address_block(addr'', size') ⊆ ac'`allocated(pm')(s'))
{-9}  OK?(ac'`free(pm')(addr')(s'))
------------------------------------------------------------------------
{1}   disjoint?(address_block(addr'', size'),
                address_block(addr', down(ac'`freed_size(pm')(addr')(s'))))
{2}   addr'' = addr' ∧ size' = down(ac'`freed_size(pm')(addr')(s'))
```

Expanding the definition of blocks_pairwise_disjoint,

Instantiating the top quantifier in -1 with the terms: addr', addr'', down(ac'`freed_size(pm')(addr')(s')), size',

Splitting conjunctions,

we get 3 subgoals:

`consistent_allocator_free.2.2.2.1:`

```
{-1}  blocks_disjoint?(addr', down(ac'`freed_size(pm')(addr')(s')), addr'', size')
{-2}  member((addr'', size'), ai')
{-3}  size' > 0
{-4}  plain_memory?(pm')
{-5}  allocator?(pm')(ac')
{-6}  pm'`states(s')
{-7}  allocator_info?(ai')
{-8}  (address_block(addr'', size') ⊆ ac'`allocated(pm')(s'))
{-9}  OK?(ac'`free(pm')(addr')(s'))
------------------------------------------------------------------------
{1}   disjoint?(address_block(addr'', size'),
                address_block(addr', down(ac'`freed_size(pm')(addr')(s'))))
{2}   addr'' = addr' ∧ size' = down(ac'`freed_size(pm')(addr')(s'))
```

Rewriting using blocks_disjoint_disjoint, matching in *,

Rewriting using disjoint_symmetric, matching in *,

This completes the proof of `consistent_allocator_free.2.2.2.1`.

`consistent_allocator_free.2.2.2.2:`

```
{-1}  member((addr'', size'), ai')
{-2}  size' > 0
{-3}  plain_memory?(pm')
{-4}  allocator?(pm')(ac')
{-5}  pm'`states(s')
{-6}  allocator_info?(ai')
{-7}  (address_block(addr'', size') ⊆ ac'`allocated(pm')(s'))
{-8}  OK?(ac'`free(pm')(addr')(s'))
------------------------------------------------------------------------
{1}   member((addr', down(ac'`freed_size(pm')(addr')(s'))),
             cons((addr', down(ac'`freed_size(pm')(addr')(s'))), ai'))
{2}   disjoint?(address_block(addr'', size'),
                address_block(addr', down(ac'`freed_size(pm')(addr')(s'))))
{3}   addr'' = addr' ∧ size' = down(ac'`freed_size(pm')(addr')(s'))
```

Expanding the definition of member,
which is trivially true.
This completes the proof of `consistent_allocator_free.2.2.2.2`.
`consistent_allocator_free.2.2.2.3`:

```
{-1}  member((addr'', size'), ai')
{-2}  size' > 0
{-3}  plain_memory?(pm')
{-4}  allocator?(pm')(ac')
{-5}  pm'`states(s')
{-6}  allocator_info?(ai')
{-7}  (address_block(addr'', size') ⊆ ac'`allocated(pm')(s'))
{-8}  OK?(ac'`free(pm')(addr')(s'))
------------------------------------------------------------------------
{1}   member((addr'', size'),
             list_remove((addr', down(ac'`freed_size(pm')(addr')(s'))),
                         cons((addr', down(ac'`freed_size(pm')(addr')(s'))), ai')))
{2}   disjoint?(address_block(addr'', size'),
                address_block(addr', down(ac'`freed_size(pm')(addr')(s'))))
{3}   addr'' = addr' ∧ size' = down(ac'`freed_size(pm')(addr')(s'))
```

Rewriting using member_list_remove, matching in *,
Expanding the definition of member,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `consistent_allocator_free.2.2.2.3`.
Q.E.D.

C.9 Proofs for Allocation_Table (allocators.pvs)

C.9.1 Allocation_Table.allocation_finite_tree

Terse proof for `allocation_finite_tree`.
`allocation_finite_tree`:

```
------------------------------------------------------------------------
{1}   ∀ (pm: (plain_memory?), at: Allocation_Table(pm)): finite_tree?(at`hierarchy)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `allocation_finite_tree`.
Q.E.D.

C.9.2 Allocation_Table.allocation_subnodes_allocated_and_free

Terse proof for `allocation_subnodes_allocated_and_free`.
`allocation_subnodes_allocated_and_free`:

```
------------------------------------------------------------------------
{1}   ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
           ac1, ac2: (at`hierarchy`nodes)):
        allocation_ok?(pm)(at, s) ∧ at`hierarchy`edges(ac1, ac2) ⊃
          (ac2`memory_pool(pm)(s) ⊆ ac1`allocated(pm)(s)) ∧
            block_is_free(ac2`memory_pool(pm)(s), at`info(ac1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocation_ok?,

Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_subnodes_allocated_and_free`.
Q.E.D.

C.9.3 Allocation_Table.allocation_subnodes_disjoint

Terse proof for `allocation_subnodes_disjoint`.
`allocation_subnodes_disjoint`:

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
          ac1, ac2, ac3: (at‘hierarchy‘nodes)):
       allocation_ok?(pm)(at, s) ∧
        at‘hierarchy‘edges(ac1, ac2) ∧ at‘hierarchy‘edges(ac1, ac3)
        ⊃ ac2 = ac3 ∨ disjoint?(ac2‘memory_pool(pm)(s), ac3‘memory_pool(pm)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocation_ok?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -9 with the terms: ac1′, ac2′, ac3′,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_subnodes_disjoint`.
Q.E.D.

C.9.4 Allocation_Table.allocation_roots_disjoint

Terse proof for `allocation_roots_disjoint`.
`allocation_roots_disjoint`:

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
          ac1, ac2: (roots(at‘hierarchy))):
       allocation_ok?(pm)(at, s) ⊃
        ac1 = ac2 ∨ disjoint?(ac1‘memory_pool(pm)(s), ac2‘memory_pool(pm)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocation_ok?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_roots_disjoint`.
Q.E.D.

C.9.5 Allocation_Table.allocation_private_memory_disjoint

Terse proof for `allocation_private_memory_disjoint`.
`allocation_private_memory_disjoint`:

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
          ac1, ac2: (at‘hierarchy‘nodes)):
       allocation_ok?(pm)(at, s) ⊃
        ac1 = ac2 ∨ disjoint?(ac1‘private_mem(pm), ac2‘private_mem(pm))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocation_ok?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_private_memory_disjoint`.
Q.E.D.

C.9.6 Allocation_Table.allocation_consistent_allocator

Terse proof for `allocation_consistent_allocator`.
`allocation_consistent_allocator:`

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at‘hierarchy‘nodes)):
       allocation_ok?(pm)(at, s) ⊃ consistent_allocator?(pm)(ac, at‘info(ac))(s)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocation_ok?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `allocation_consistent_allocator`.
Q.E.D.

C.9.7 Allocation_Table.allocation_allocated_allocation_point

Terse proof for `allocation_allocated_allocation_point`.
`allocation_allocated_allocation_point:`

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at‘hierarchy‘nodes),
          addr: Address, size: posnat):
       allocation_ok?(pm)(at, s) ∧ member((addr, size), at‘info(ac)) ⊃
        (address_block(addr, size) ⊆ ac‘allocated(pm)(s))
```

Repeatedly Skolemizing and flattening,
Using lemma allocation_consistent_allocator,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of consistent_allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_allocated_allocation_point`.
Q.E.D.

C.9.8 Allocation_Table.allocation_allocation_point_in_memory_pool

Terse proof for `allocation_allocation_point_in_memory_pool`.
`allocation_allocation_point_in_memory_pool:`

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at‘hierarchy‘nodes),
          addr: Address, size: posnat):
       pm‘states(s) ∧ allocation_ok?(pm)(at, s) ∧ member((addr, size), at‘info(ac)) ⊃
        (address_block(addr, size) ⊆ ac‘memory_pool(pm)(s))
```

Repeatedly Skolemizing and flattening,
Using lemma allocation_allocated_allocation_point,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_subset_allocated_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on subset_transitive,
This completes the proof of `allocation_allocation_point_in_memory_pool`.
Q.E.D.

C.9.9 Allocation_Table.allocators_different_in_hierachy

Terse proof for `allocators_different_in_hierachy`.

`allocators_different_in_hierachy`:

```
{1}  ∀ (pm: (plain_memory?), at: Allocation_Table(pm), ac1, ac2: (at‘hierarchy‘nodes)):
       at‘hierarchy‘edges(ac1, ac2) ⊃ ¬ ac1 = ac2
```

Repeatedly Skolemizing and flattening,
Using lemma different_nodes_on_edge[(allocator?(pm′))],
Simplifying, rewriting, and recording with decision procedures,
Adding type constraints for at!1‘hierarchy,
Expanding the definition of finite_tree?,
Expanding the definition of tree?,
which is trivially true.
This completes the proof of `allocators_different_in_hierachy`.
Q.E.D.

C.9.10 Allocation_Table.uppath_memory_pool

Terse proof for `uppath_memory_pool`.

`uppath_memory_pool`:

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), p: (path?(at‘hierarchy))):
       allocation_ok?(pm)(at, s) ∧ pm‘states(s) ⊃
         (path_end(p)‘memory_pool(pm)(s) ⊆ path_start(p)‘memory_pool(pm)(s))
```

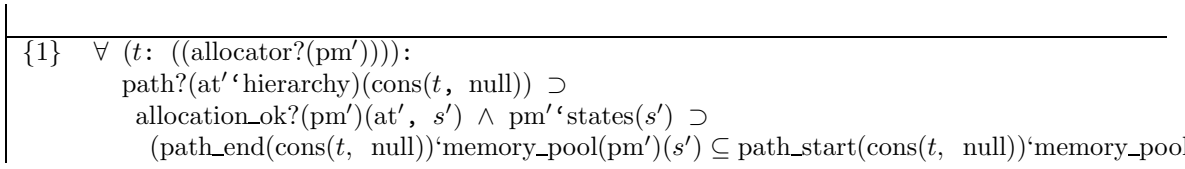
For the top quantifier in 1, we introduce Skolem constants: (pm′ s' at′ _),
Inducting on p on formula 1 using induction scheme path_list_induction[(allocator?(pm′))],
we get 3 subgoals:

`uppath_memory_pool.1`:

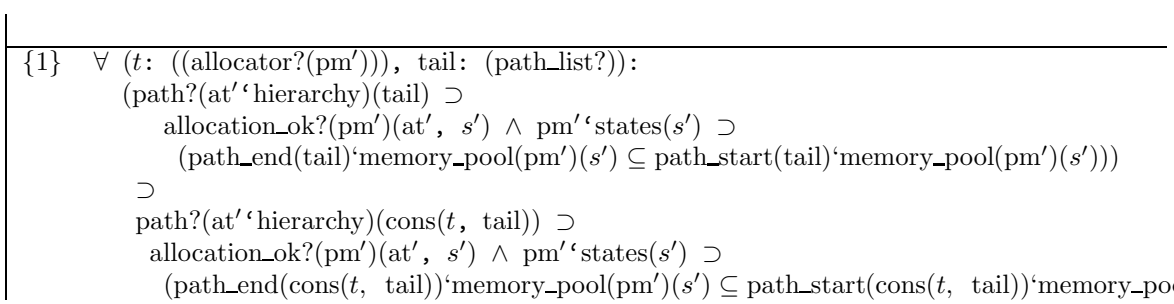
```
{-1}  allocation_ok?(pm′)(at′, s′)
{-2}  pm′‘states(s′)
------------------------------------------------------------------------
{1}   path?(at′‘hierarchy)(p′)
{2}   (path_end(p′)‘memory_pool(pm′)(s′) ⊆ path_start(p′)‘memory_pool(pm′)(s′))
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `uppath_memory_pool.1`.

`uppath_memory_pool.2`:



Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_start path_end length nth subset_reflexive
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `uppath_memory_pool.2`.
`uppath_memory_pool.3`:



Repeatedly Skolemizing and flattening,
Using lemma path_cdr,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: path_start nth length path_list? path_end
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of path?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -7 with the terms: 0,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocation_subnodes_allocated_and_free,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Hiding formulas: -2,
Forward chaining on subset_transitive,
Hiding formulas: -2, -7,
Using lemma allocator_subset_allocated_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on subset_transitive,
This completes the proof of `uppath_memory_pool.3`.
Q.E.D.

C.9.11 Allocation_Table.uppath_memory_pool_allocated_and_free_TCC1

Terse proof for `uppath_memory_pool_allocated_and_free_TCC1`.

uppath_memory_pool_allocated_and_free_TCC1:

{1} ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
p: (path?[((allocator?(pm)))](at‘hierarchy))):
(path_end(p)‘memory_pool(pm)(s) ⊆ path_start(p)‘allocated(pm)(s)) ∧
pm‘states(s) ∧ allocation_ok?(pm)(at, s) ∧ length(p) ≥ 2
⊃ at‘hierarchy‘nodes(path_start[((allocator?(pm)))](p))

Repeatedly Skolemizing and flattening,
Rewriting using nodes_path_start, matching in *,
This completes the proof of uppath_memory_pool_allocated_and_free_TCC1.
Q.E.D.

C.9.12 Allocation_Table.uppath_memory_pool_allocated_and_free

Terse proof for uppath_memory_pool_allocated_and_free.

uppath_memory_pool_allocated_and_free:

{1} ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), p: (path?(at‘hierarchy))):
pm‘states(s) ∧ allocation_ok?(pm)(at, s) ∧ length(p) ≥ 2 ⊃
(path_end(p)‘memory_pool(pm)(s) ⊆ path_start(p)‘allocated(pm)(s)) ∧
block_is_free(path_end(p)‘memory_pool(pm)(s), at‘info(path_start(p)))

Repeatedly Skolemizing and flattening,
Using lemma path_cdr,
Installing automatic rewrites from: path_list? length_cdr path_end path_start nth subset_reflexive
Simplifying, rewriting, and recording with decision procedures,
Using lemma uppath_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of path?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -7 with the terms: 0,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocation_subnodes_allocated_and_free,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Forward chaining on subset_transitive,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -1, -2,
Expanding the definition of block_is_free,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Using lemma disjoint_mono,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of uppath_memory_pool_allocated_and_free.
Q.E.D.

C.9.13 Allocation_Table.allocation_memory_pools

Terse proof for allocation_memory_pools.

`allocation_memory_pools:`

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
          ac1, ac2: (at‘hierarchy‘nodes)):
        pm‘states(s) ∧ allocation_ok?(pm)(at, s) ⊃
         ac1 = ac2 ∨
          disjoint?(ac1‘memory_pool(pm)(s), ac2‘memory_pool(pm)(s)) ∨
           ((ac1‘memory_pool(pm)(s) ⊆ ac2‘allocated(pm)(s)) ∧
              block_is_free(ac1‘memory_pool(pm)(s), at‘info(ac2)))
            ∨
            ((ac2‘memory_pool(pm)(s) ⊆ ac1‘allocated(pm)(s)) ∧
               block_is_free(ac2‘memory_pool(pm)(s), at‘info(ac1)))
```

Repeatedly Skolemizing and flattening,

Using lemma symmetric_node_pair_distinction[(allocator?(pm′))],

Splitting conjunctions,

we get 7 subgoals:

`allocation_memory_pools.1:`

```
{-1}  ∀ (n1_1, n2_1: (at′‘hierarchy‘nodes)):
        n1_1 = n2_1 ∨
         disjoint?(n1_1‘memory_pool(pm′)(s′), n2_1‘memory_pool(pm′)(s′)) ∨
          ((n1_1‘memory_pool(pm′)(s′) ⊆ n2_1‘allocated(pm′)(s′)) ∧
             block_is_free(n1_1‘memory_pool(pm′)(s′), at′‘info(n2_1)))
           ∨
           ((n2_1‘memory_pool(pm′)(s′) ⊆ n1_1‘allocated(pm′)(s′)) ∧
              block_is_free(n2_1‘memory_pool(pm′)(s′), at′‘info(n1_1)))
{-2}  plain_memory?(pm′)
{-3}  allocator?(pm′)(ac1′)
{-4}  at′‘hierarchy‘nodes(ac1′)
{-5}  allocator?(pm′)(ac2′)
{-6}  at′‘hierarchy‘nodes(ac2′)
{-7}  pm′‘states(s′)
{-8}  allocation_ok?(pm′)(at′, s′)
------------------------------------------------------------
{1}   ac1′ = ac2′
{2}   disjoint?(ac1′‘memory_pool(pm′)(s′), ac2′‘memory_pool(pm′)(s′))
{3}   ((ac1′‘memory_pool(pm′)(s′) ⊆ ac2′‘allocated(pm′)(s′)) ∧
         block_is_free(ac1′‘memory_pool(pm′)(s′), at′‘info(ac2′)))
{4}   ((ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(s′)) ∧
         block_is_free(ac2′‘memory_pool(pm′)(s′), at′‘info(ac1′)))
```

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `allocation_memory_pools.1`.

`allocation_memory_pools.2:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac1′)
{-3} at′‘hierarchy‘nodes(ac1′)
{-4} allocator?(pm′)(ac2′)
{-5} at′‘hierarchy‘nodes(ac2′)
{-6} pm′‘states(s′)
{-7} allocation_ok?(pm′)(at′, s′)

{1} TRUE
{2} ac1′ = ac2′
{3} disjoint?(ac1′‘memory_pool(pm′)(s′), ac2′‘memory_pool(pm′)(s′))
{4} ((ac1′‘memory_pool(pm′)(s′) ⊆ ac2′‘allocated(pm′)(s′)) ∧
block_is_free(ac1′‘memory_pool(pm′)(s′), at′‘info(ac2′)))
{5} ((ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(s′)) ∧
block_is_free(ac2′‘memory_pool(pm′)(s′), at′‘info(ac1′)))

which is trivially true.

This completes the proof of `allocation_memory_pools.2`.

`allocation_memory_pools.3:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac1′)
{-3} at′‘hierarchy‘nodes(ac1′)
{-4} allocator?(pm′)(ac2′)
{-5} at′‘hierarchy‘nodes(ac2′)
{-6} pm′‘states(s′)
{-7} allocation_ok?(pm′)(at′, s′)

{1} symmetric?(λ (n1, n2: (at′‘hierarchy‘nodes)):
n1 = n2 ∨
disjoint?(n1‘memory_pool(pm′)(s′), n2‘memory_pool(pm′)(s′)) ∨
((n1‘memory_pool(pm′)(s′) ⊆ n2‘allocated(pm′)(s′)) ∧
block_is_free(n1‘memory_pool(pm′)(s′), at′‘info(n2)))
∨
((n2‘memory_pool(pm′)(s′) ⊆ n1‘allocated(pm′)(s′)) ∧
block_is_free(n2‘memory_pool(pm′)(s′), at′‘info(n1))))
{2} ac1′ = ac2′
{3} disjoint?(ac1′‘memory_pool(pm′)(s′), ac2′‘memory_pool(pm′)(s′))
{4} ((ac1′‘memory_pool(pm′)(s′) ⊆ ac2′‘allocated(pm′)(s′)) ∧
block_is_free(ac1′‘memory_pool(pm′)(s′), at′‘info(ac2′)))
{5} ((ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(s′)) ∧
block_is_free(ac2′‘memory_pool(pm′)(s′), at′‘info(ac1′)))

Keeping 1 and hiding *,

Expanding the definition of symmetric?,

Repeatedly Skolemizing and flattening,

Using lemma disjoint_symmetric,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `allocation_memory_pools.3`.

`allocation_memory_pools.4:`

{-1}	plain_memory?(pm′)
{-2}	allocator?(pm′)(ac1′)
{-3}	at′‘hierarchy‘nodes(ac1′)
{-4}	allocator?(pm′)(ac2′)
{-5}	at′‘hierarchy‘nodes(ac2′)
{-6}	pm′‘states(s')
{-7}	allocation_ok?(pm′)(at′, s')
{1}	∀ (n: (at′‘hierarchy‘nodes)): TRUE
{2}	ac1′ = ac2′
{3}	disjoint?(ac1′‘memory_pool(pm′)(s'), ac2′‘memory_pool(pm′)(s'))
{4}	((ac1′‘memory_pool(pm′)(s') ⊆ ac2′‘allocated(pm′)(s')) ∧ block_is_free(ac1′‘memory_pool(pm′)(s'), at′‘info(ac2′)))
{5}	((ac2′‘memory_pool(pm′)(s') ⊆ ac1′‘allocated(pm′)(s')) ∧ block_is_free(ac2′‘memory_pool(pm′)(s'), at′‘info(ac1′)))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_memory_pools.4`.

`allocation_memory_pools.5:`

{-1}	plain_memory?(pm′)
{-2}	allocator?(pm′)(ac1′)
{-3}	at′‘hierarchy‘nodes(ac1′)
{-4}	allocator?(pm′)(ac2′)
{-5}	at′‘hierarchy‘nodes(ac2′)
{-6}	pm′‘states(s')
{-7}	allocation_ok?(pm′)(at′, s')
{1}	∀ (n1_1, n2_1: (at′‘hierarchy‘nodes), p: (path?(at′‘hierarchy))): ¬ n1_1 = n2_1 ∧ path_start(p) = n2_1 ∧ path_end(p) = n1_1 ∧ length(p) ≥ 2 ⊃ n1_1 = n2_1 ∨ disjoint?(n1_1‘memory_pool(pm′)(s'), n2_1‘memory_pool(pm′)(s')) ∨ ((n1_1‘memory_pool(pm′)(s') ⊆ n2_1‘allocated(pm′)(s')) ∧ block_is_free(n1_1‘memory_pool(pm′)(s'), at′‘info(n2_1))) ∨ ((n2_1‘memory_pool(pm′)(s') ⊆ n1_1‘allocated(pm′)(s')) ∧ block_is_free(n2_1‘memory_pool(pm′)(s'), at′‘info(n1_1)))
{2}	ac1′ = ac2′
{3}	disjoint?(ac1′‘memory_pool(pm′)(s'), ac2′‘memory_pool(pm′)(s'))
{4}	((ac1′‘memory_pool(pm′)(s') ⊆ ac2′‘allocated(pm′)(s')) ∧ block_is_free(ac1′‘memory_pool(pm′)(s'), at′‘info(ac2′)))
{5}	((ac2′‘memory_pool(pm′)(s') ⊆ ac1′‘allocated(pm′)(s')) ∧ block_is_free(ac2′‘memory_pool(pm′)(s'), at′‘info(ac1′)))

Keeping (-6 -7 1) and hiding *,

Repeatedly Skolemizing and flattening,

Using lemma uppath_memory_pool_allocated_and_free,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `allocation_memory_pools.5`.

`allocation_memory_pools.6:`

```
{-1}  plain_memory?(pm′)
{-2}  allocator?(pm′)(ac1′)
{-3}  at′‘hierarchy‘nodes(ac1′)
{-4}  allocator?(pm′)(ac2′)
{-5}  at′‘hierarchy‘nodes(ac2′)
{-6}  pm′‘states(s′)
{-7}  allocation_ok?(pm′)(at′, s′)
------------------------------------------------------------------------
{1}   ∀ (n1_1, n2_1: (at′‘hierarchy‘nodes), p1, p2: (path?(at′‘hierarchy))):
        ¬ n1_1 = n2_1 ∧
         root_path(at′‘hierarchy, n1_1)(p1) ∧
          root_path(at′‘hierarchy, n2_1)(p2) ∧ ¬ path_start(p1) = path_start(p2)
         ⊃
         n1_1 = n2_1 ∨
          disjoint?(n1_1‘memory_pool(pm′)(s′), n2_1‘memory_pool(pm′)(s′)) ∨
           ((n1_1‘memory_pool(pm′)(s′) ⊆ n2_1‘allocated(pm′)(s′)) ∧
              block_is_free(n1_1‘memory_pool(pm′)(s′), at′‘info(n2_1)))
            ∨
            ((n2_1‘memory_pool(pm′)(s′) ⊆ n1_1‘allocated(pm′)(s′)) ∧
               block_is_free(n2_1‘memory_pool(pm′)(s′), at′‘info(n1_1)))
{2}   ac1′ = ac2′
{3}   disjoint?(ac1′‘memory_pool(pm′)(s′), ac2′‘memory_pool(pm′)(s′))
{4}   ((ac1′‘memory_pool(pm′)(s′) ⊆ ac2′‘allocated(pm′)(s′)) ∧
         block_is_free(ac1′‘memory_pool(pm′)(s′), at′‘info(ac2′)))
{5}   ((ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(s′)) ∧
         block_is_free(ac2′‘memory_pool(pm′)(s′), at′‘info(ac1′)))
```

Keeping (-6 -7 1) and hiding *,

Repeatedly Skolemizing and flattening,

Hiding formulas: 5, 6,

Using lemma uppath_memory_pool,

Using lemma uppath_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Using lemma allocation_roots_disjoint,

Simplifying, rewriting, and recording with decision procedures,

Forward chaining on disjoint_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_memory_pools.6`.

`allocation_memory_pools.7:`

```
{-1}  plain_memory?(pm')
{-2}  allocator?(pm')(ac1')
{-3}  at'`hierarchy`nodes(ac1')
{-4}  allocator?(pm')(ac2')
{-5}  at'`hierarchy`nodes(ac2')
{-6}  pm'`states(s')
{-7}  allocation_ok?(pm')(at', s')
------------------------------------------------------------------------
{1}   ∀ (n1_1, n2_1, n3: (at'`hierarchy`nodes), p1, p2: (path?(at'`hierarchy))):
        ¬ n1_1 = n2_1 ∧
         ¬ n1_1 = n3 ∧
          ¬ n2_1 = n3 ∧
           at'`hierarchy`edges(n3, path_start(p1)) ∧
            at'`hierarchy`edges(n3, path_start(p2)) ∧
             path_end(p1) = n1_1 ∧ path_end(p2) = n2_1 ∧ ¬ path_start(p1) = path_start(p2)
          ⊃
          n1_1 = n2_1 ∨
           disjoint?(n1_1`memory_pool(pm')(s'), n2_1`memory_pool(pm')(s')) ∨
            ((n1_1`memory_pool(pm')(s') ⊆ n2_1`allocated(pm')(s')) ∧
               block_is_free(n1_1`memory_pool(pm')(s'), at'`info(n2_1)))
             ∨
             ((n2_1`memory_pool(pm')(s') ⊆ n1_1`allocated(pm')(s')) ∧
                block_is_free(n2_1`memory_pool(pm')(s'), at'`info(n1_1)))
{2}   ac1' = ac2'
{3}   disjoint?(ac1'`memory_pool(pm')(s'), ac2'`memory_pool(pm')(s'))
{4}   ((ac1'`memory_pool(pm')(s') ⊆ ac2'`allocated(pm')(s')) ∧
         block_is_free(ac1'`memory_pool(pm')(s'), at'`info(ac2')))
{5}   ((ac2'`memory_pool(pm')(s') ⊆ ac1'`allocated(pm')(s')) ∧
         block_is_free(ac2'`memory_pool(pm')(s'), at'`info(ac1')))
```

Keeping (-6 -7 1) and hiding *,
Repeatedly Skolemizing and flattening,
Hiding formulas: 7, 8,
Using lemma uppath_memory_pool,
Using lemma uppath_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocation_subnodes_disjoint,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on disjoint_mono,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_memory_pools.7`.
Q.E.D.

C.9.14 Allocation_Table.allocation_info_TCC1

Terse proof for `allocation_info_TCC1`.

`allocation_info_TCC1:`

```
------------------------------------------------------------------------
{1}   ∀ (pm: (plain_memory?[State]), at: Allocation_Table(pm)):
        is_finite[(at`hierarchy`nodes)]
             (restrict[((allocator?(pm))), (at`hierarchy`nodes), boolean]
                  (at`hierarchy`nodes))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for at!1‘hierarchy,
Expanding the definition of finite_tree?,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of finite?,
Hiding formulas: -2, -3,
Expanding the definition of restrict,
Expanding the definition of is_finite,
which is trivially true.
This completes the proof of `allocation_info_TCC1`.
Q.E.D.

C.9.15 Allocation_Table.member_allocation_info

Terse proof for `member_allocation_info`.
`member_allocation_info:`

```
{1}  ∀ (pm: (plain_memory?), at: Allocation_Table(pm), addr: Address, size: posnat):
       member((addr, size), allocation_info(pm)(at)) ⊃
         (∃ (ac: (at‘hierarchy‘nodes)): member((addr, size), at‘info(ac)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocation_info,
Using lemma member_flatten[[Address, posnat]],
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -4,
Repeatedly Skolemizing and flattening,
Using lemma member_map,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -2,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `member_allocation_info`.
Q.E.D.

C.9.16 Allocation_Table.allocation_consistent

Terse proof for `allocation_consistent`.
`allocation_consistent:`

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm)):
       pm‘states(s) ∧ allocation_ok?(pm)(at, s) ⊃
         allocator_info?(allocation_info(pm)(at))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator_info?,
Expanding the definition of blocks_pairwise_disjoint,
Repeatedly Skolemizing and flattening,
Rewriting using member_list_remove, matching in *,
Splitting conjunctions,
we get 2 subgoals:

`allocation_consistent.1:`

{-1}	$\text{size}' > 0$
{-2}	$\text{size}'' > 0$
{-3}	$\text{plain\_memory?}(\text{pm}')$
{-4}	$\text{pm}'\text{`states}(s')$
{-5}	$\text{allocation\_ok?}(\text{pm}')(\text{at}', s')$
{-6}	$\text{member}((\text{addr}', \text{size}'), \text{allocation\_info}(\text{pm}')(\text{at}'))$
{1}	TRUE
{2}	$\text{blocks\_disjoint?}(\text{addr}', \text{size}', \text{addr}'', \text{size}'')$

which is trivially true.
This completes the proof of `allocation_consistent.1`.
`allocation_consistent.2:`

{-1}	$\neg\ (\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}') \wedge$ $\text{member}((\text{addr}'', \text{size}''), \text{allocation\_info}(\text{pm}')(\text{at}'))$
{-2}	$\text{size}' > 0$
{-3}	$\text{size}'' > 0$
{-4}	$\text{plain\_memory?}(\text{pm}')$
{-5}	$\text{pm}'\text{`states}(s')$
{-6}	$\text{allocation\_ok?}(\text{pm}')(\text{at}', s')$
{-7}	$\text{member}((\text{addr}', \text{size}'), \text{allocation\_info}(\text{pm}')(\text{at}'))$
{1}	$\text{blocks\_disjoint?}(\text{addr}', \text{size}', \text{addr}'', \text{size}'')$

Applying disjunctive simplification to flatten sequent,
Forward chaining on member_allocation_info,
Forward chaining on member_allocation_info,
Hiding formulas: -3, -9,
Repeatedly Skolemizing and flattening,
Using lemma allocation_memory_pools,
Simplifying, rewriting, and recording with decision procedures,
Splitting conjunctions,
we get 4 subgoals:
`allocation_consistent.2.1:`

{-1}	$\text{ac}' = \text{ac}''$
{-2}	$\text{allocator?}(\text{pm}')(\text{ac}'')$
{-3}	$\text{at}'\text{`hierarchy`nodes}(\text{ac}'')$
{-4}	$\text{allocator?}(\text{pm}')(\text{ac}')$
{-5}	$\text{at}'\text{`hierarchy`nodes}(\text{ac}')$
{-6}	$\text{member}((\text{addr}', \text{size}'), \text{at}'\text{`info}(\text{ac}'))$
{-7}	$\text{member}((\text{addr}'', \text{size}''), \text{at}'\text{`info}(\text{ac}''))$
{-8}	$\text{size}' > 0$
{-9}	$\text{size}'' > 0$
{-10}	$\text{plain\_memory?}(\text{pm}')$
{-11}	$\text{pm}'\text{`states}(s')$
{-12}	$\text{allocation\_ok?}(\text{pm}')(\text{at}', s')$
{1}	$\text{addr}'' = \text{addr}' \wedge \text{size}'' = \text{size}'$
{2}	$\text{blocks\_disjoint?}(\text{addr}', \text{size}', \text{addr}'', \text{size}'')$

Using lemma allocation_consistent_allocator,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of consistent_allocator?,
Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2,
Expanding the definition of allocator_info?,
Expanding the definition of blocks_pairwise_disjoint,
Instantiating quantified variables,
Rewriting using member_list_remove, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `allocation_consistent.2.1`.
`allocation_consistent.2.2`:

```
{-1}  disjoint?(ac'`memory_pool(pm')(s'), ac''`memory_pool(pm')(s'))
{-2}  allocator?(pm')(ac'')
{-3}  at'`hierarchy`nodes(ac'')
{-4}  allocator?(pm')(ac')
{-5}  at'`hierarchy`nodes(ac')
{-6}  member((addr', size'), at'`info(ac'))
{-7}  member((addr'', size''), at'`info(ac''))
{-8}  size' > 0
{-9}  size'' > 0
{-10} plain_memory?(pm')
{-11} pm'`states(s')
{-12} allocation_ok?(pm')(at', s')
------------------------------------------------------------
 {1}  addr'' = addr' ∧ size'' = size'
 {2}  blocks_disjoint?(addr', size', addr'', size'')
```

Rewriting using blocks_disjoint_disjoint, matching in *,
Using lemma allocation_allocation_point_in_memory_pool,
Using lemma allocation_allocation_point_in_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on disjoint_mono,
This completes the proof of `allocation_consistent.2.2`.
`allocation_consistent.2.3`:

```
{-1}  ((ac'`memory_pool(pm')(s') ⊆ ac''`allocated(pm')(s')) ∧
          block_is_free(ac'`memory_pool(pm')(s'), at'`info(ac'')))
{-2}  allocator?(pm')(ac'')
{-3}  at'`hierarchy`nodes(ac'')
{-4}  allocator?(pm')(ac')
{-5}  at'`hierarchy`nodes(ac')
{-6}  member((addr', size'), at'`info(ac'))
{-7}  member((addr'', size''), at'`info(ac''))
{-8}  size' > 0
{-9}  size'' > 0
{-10} plain_memory?(pm')
{-11} pm'`states(s')
{-12} allocation_ok?(pm')(at', s')
------------------------------------------------------------
 {1}  addr'' = addr' ∧ size'' = size'
 {2}  blocks_disjoint?(addr', size', addr'', size'')
```

Applying disjunctive simplification to flatten sequent,
Rewriting using blocks_disjoint_disjoint, matching in *,
Using lemma allocation_allocation_point_in_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of block_is_free,
Instantiating the top quantifier in -3 with the terms: addr'', size'',

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Using lemma disjoint_mono,

Installing automatic rewrites from: subset_reflexive

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_consistent.2.3`.

`allocation_consistent.2.4`:

{-1}	((ac″‘memory_pool(pm′)(s′) ⊆ ac′‘allocated(pm′)(s′)) ∧ block_is_free(ac″‘memory_pool(pm′)(s′), at′‘info(ac′)))
{-2}	allocator?(pm′)(ac″)
{-3}	at′‘hierarchy‘nodes(ac″)
{-4}	allocator?(pm′)(ac′)
{-5}	at′‘hierarchy‘nodes(ac′)
{-6}	member((addr′, size′), at′‘info(ac′))
{-7}	member((addr″, size″), at′‘info(ac″))
{-8}	size′ > 0
{-9}	size″ > 0
{-10}	plain_memory?(pm′)
{-11}	pm′‘states(s′)
{-12}	allocation_ok?(pm′)(at′, s′)
{1}	addr″ = addr′ ∧ size″ = size′
{2}	blocks_disjoint?(addr′, size′, addr″, size″)

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -1,

Rewriting using blocks_disjoint_disjoint, matching in *,

Using lemma allocation_allocation_point_in_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of block_is_free,

Instantiating the top quantifier in -2 with the terms: addr′, size′,

Simplifying, rewriting, and recording with decision procedures,

Using lemma disjoint_mono,

Installing automatic rewrites from: subset_reflexive

Simplifying, rewriting, and recording with decision procedures,

Rewriting using disjoint_symmetric, matching in *,

This completes the proof of `allocation_consistent.2.4`.

Q.E.D.

C.9.17 Allocation_Table.allocation_alloc_other_private_memory_TCC1

Terse proof for `allocation_alloc_other_private_memory_TCC1`.

allocation_alloc_other_private_memory_TCC1:

{1} ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
ac1, ac2: (at‘hierarchy‘nodes), size: posnat):
pm‘states(s) ∧
allocation_ok?(pm)(at, s) ∧
OK?(ac1‘alloc(pm)(size)(s)) ∧
¬ data(ac1‘alloc(pm)(size)(s)) = null_address ∧ ¬ ac1 = ac2
⊃
OK?[State, Address](ac1‘alloc(pm)(size)(s)) ∨
Exception?[State, Address](ac1‘alloc(pm)(size)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of allocation_alloc_other_private_memory_TCC1.
Q.E.D.

C.9.18 Allocation_Table.allocation_alloc_other_private_memory_TCC2

Terse proof for allocation_alloc_other_private_memory_TCC2.
allocation_alloc_other_private_memory_TCC2:

{1} ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
ac1, ac2: (at‘hierarchy‘nodes), size: posnat):
pm‘states(s) ∧
allocation_ok?(pm)(at, s) ∧
OK?(ac1‘alloc(pm)(size)(s)) ∧
¬ data(ac1‘alloc(pm)(size)(s)) = null_address ∧ ¬ ac1 = ac2
⊃ pm‘states(state[State, Address](ac1‘alloc(pm)(size)(s)))

Repeatedly Skolemizing and flattening,
Rewriting using allocator_next_state_alloc, matching in *,
Expanding the definition of has_next_state,
which is trivially true.
This completes the proof of allocation_alloc_other_private_memory_TCC2.
Q.E.D.

C.9.19 Allocation_Table.allocation_alloc_other_private_memory

Terse proof for allocation_alloc_other_private_memory.
allocation_alloc_other_private_memory:

{1} ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm),
ac1, ac2: (at‘hierarchy‘nodes), size: posnat):
pm‘states(s) ∧
allocation_ok?(pm)(at, s) ∧
OK?(ac1‘alloc(pm)(size)(s)) ∧
¬ data(ac1‘alloc(pm)(size)(s)) = null_address ∧ ¬ ac1 = ac2
⊃ stays_unchanged(pm)(s, state(ac1‘alloc(pm)(size)(s)), ac2‘private_mem(pm))

Repeatedly Skolemizing and flattening,

Using lemma allocator_only_changes_alloc,
Simplifying, rewriting, and recording with decision procedures,
Using lemma stays_unchanged_only_changes_disjoint,
Installing automatic rewrites from: allocator_next_state_alloc allocator_subset_private_memory_plain_memory has_next_state
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocation_private_memory_disjoint,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_alloc_other_private_memory`.
Q.E.D.

C.9.20 Allocation_Table.allocation_alloc_TCC1

Terse proof for `allocation_alloc_TCC1`.
`allocation_alloc_TCC1`:

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at‘hierarchy‘nodes),
         size: posnat):
       pm‘states(s) ∧
        allocation_ok?(pm)(at, s) ∧
         OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
        ⊃
        OK?[State, Address](ac‘alloc(pm)(size)(s)) ∨
         Exception?[State, Address](ac‘alloc(pm)(size)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `allocation_alloc_TCC1`.
Q.E.D.

C.9.21 Allocation_Table.allocation_alloc

Terse proof for `allocation_alloc`.
`allocation_alloc`:

```
{1}  ∀ (pm: (plain_memory?), s: State, at: Allocation_Table(pm), ac: (at‘hierarchy‘nodes),
         size: posnat):
       pm‘states(s) ∧
        allocation_ok?(pm)(at, s) ∧
         OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
        ⊃
        allocation_ok?(pm)
                      (at
                           WITH [‘info(ac)
                                   := cons((data(ac‘alloc(pm)(size)(s)), size),
                                           at‘info(ac))],
                        state(ac‘alloc(pm)(size)(s)))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: allocator_next_state_alloc has_next_state subset_bigger_union_left allocator_memory_pool_alloc allocator_allocated_alloc subset_equal
Expanding the definition of allocation_ok?,
Applying propositional simplification,

we get 5 subgoals:
`allocation_alloc.1:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac′)
{-3} at′‘hierarchy‘nodes(ac′)
{-4} size′ > 0
{-5} pm′‘states(s′)
{-6} allocation_ok?(pm′)(at′, s′)
{-7} OK?(ac′‘alloc(pm′)(size′)(s′))

{1} ∀ (ac1, ac2: (at′‘hierarchy‘nodes)):
at′‘hierarchy‘edges(ac1, ac2) ⊃
(ac2‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))) ⊆ ac1‘allocated(pm′)(state(ac′‘alloc(pm′)(size′)(s′)))
∧
block_is_free(ac2‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))),
at′‘info
WITH [(ac′)
:= cons((data(ac′‘alloc(pm′)(size′)(s′)), size′),
at′‘info(ac′))]
(ac1))
{2} data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Repeatedly Skolemizing and flattening,
Using lemma allocation_subnodes_allocated_and_free,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Case splitting on ac!1 = ac2!1,
we get 2 subgoals:
`allocation_alloc.1.1:`

{-1} ac′ = ac2′
{-2} (ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(s′))
{-3} block_is_free(ac2′‘memory_pool(pm′)(s′), at′‘info(ac1′))
{-4} allocator?(pm′)(ac1′)
{-5} at′‘hierarchy‘nodes(ac1′)
{-6} allocator?(pm′)(ac2′)
{-7} at′‘hierarchy‘nodes(ac2′)
{-8} at′‘hierarchy‘edges(ac1′, ac2′)
{-9} plain_memory?(pm′)
{-10} allocator?(pm′)(ac′)
{-11} at′‘hierarchy‘nodes(ac′)
{-12} size′ > 0
{-13} pm′‘states(s′)
{-14} allocation_ok?(pm′)(at′, s′)
{-15} OK?(ac′‘alloc(pm′)(size′)(s′))

{1} (ac2′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))) ⊆ ac1′‘allocated(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
∧
block_is_free(ac2′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))),
at′‘info
WITH [(ac′)
:= cons((data(ac′‘alloc(pm′)(size′)(s′)), size′),
at′‘info(ac′))]
(ac1′))
{2} data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocators_different_in_hierachy,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_allocated,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.1.1`.

`allocation_alloc.1.2`:

```
{-1}  (ac2'`memory_pool(pm')(s') ⊆ ac1'`allocated(pm')(s'))
{-2}  block_is_free(ac2'`memory_pool(pm')(s'), at'`info(ac1'))
{-3}  allocator?(pm')(ac1')
{-4}  at'`hierarchy`nodes(ac1')
{-5}  allocator?(pm')(ac2')
{-6}  at'`hierarchy`nodes(ac2')
{-7}  at'`hierarchy`edges(ac1', ac2')
{-8}  plain_memory?(pm')
{-9}  allocator?(pm')(ac')
{-10} at'`hierarchy`nodes(ac')
{-11} size' > 0
{-12} pm'`states(s')
{-13} allocation_ok?(pm')(at', s')
{-14} OK?(ac'`alloc(pm')(size')(s'))
------------------------------------------------------------------------
 {1}  ac' = ac2'
 {2}  (ac2'`memory_pool(pm')(state(ac'`alloc(pm')(size')(s'))) ⊆ ac1'`allocated(pm')(state(ac'`alloc(pm'
       ∧
      block_is_free(ac2'`memory_pool(pm')(state(ac'`alloc(pm')(size')(s'))),
                    at'`info
                      WITH [(ac')
                              := cons((data(ac'`alloc(pm')(size')(s')), size'),
                                      at'`info(ac'))]
                           (ac1'))
 {3}  data(ac'`alloc(pm')(size')(s')) = null_address
```

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Case splitting on ac1!1 = ac!1,

we get 2 subgoals:

`allocation_alloc.1.2.1:`

```
  {-1}   ac1′ = ac′
  {-2}   stays_unchanged(pm′)
                          (s′, state(ac′`alloc(pm′)(size′)(s′)), ac2′`private_mem(pm′))
  {-3}   (ac2′`memory_pool(pm′)(s′) ⊆ ac1′`allocated(pm′)(s′))
  {-4}   block_is_free(ac2′`memory_pool(pm′)(s′), at′`info(ac1′))
  {-5}   allocator?(pm′)(ac1′)
  {-6}   at′`hierarchy`nodes(ac1′)
  {-7}   allocator?(pm′)(ac2′)
  {-8}   at′`hierarchy`nodes(ac2′)
  {-9}   at′`hierarchy`edges(ac1′, ac2′)
  {-10}  plain_memory?(pm′)
  {-11}  allocator?(pm′)(ac′)
  {-12}  at′`hierarchy`nodes(ac′)
  {-13}  size′ > 0
  {-14}  pm′`states(s′)
  {-15}  allocation_ok?(pm′)(at′, s′)
  {-16}  OK?(ac′`alloc(pm′)(size′)(s′))
  ------------------------------------------------------------------------
  {1}    ac′ = ac2′
  {2}    (ac2′`memory_pool(pm′)(s′) ⊆ ac1′`allocated(pm′)(state(ac′`alloc(pm′)(size′)(s′))))
           ∧
         block_is_free(ac2′`memory_pool(pm′)(s′),
                       at′`info
                         WITH [(ac′)
                                 := cons((data(ac′`alloc(pm′)(size′)(s′)), size′),
                                         at′`info(ac′))]
                              (ac1′))
  {3}    data(ac′`alloc(pm′)(size′)(s′)) = null_address
```

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using block_is_free_add, matching in *,

Using lemma allocator_alloc_disjoint,

Simplifying, rewriting, and recording with decision procedures,

Using lemma disjoint_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.1.2.1`.

`allocation_alloc.1.2.2:`

```
{-1}  stays_unchanged(pm′)
                        (s′, state(ac′‘alloc(pm′)(size′)(s′)), ac2′‘private_mem(pm′))
{-2}  (ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(s′))
{-3}  block_is_free(ac2′‘memory_pool(pm′)(s′), at′‘info(ac1′))
{-4}  allocator?(pm′)(ac1′)
{-5}  at′‘hierarchy‘nodes(ac1′)
{-6}  allocator?(pm′)(ac2′)
{-7}  at′‘hierarchy‘nodes(ac2′)
{-8}  at′‘hierarchy‘edges(ac1′, ac2′)
{-9}  plain_memory?(pm′)
{-10} allocator?(pm′)(ac′)
{-11} at′‘hierarchy‘nodes(ac′)
{-12} size′ > 0
{-13} pm′‘states(s′)
{-14} allocation_ok?(pm′)(at′, s′)
{-15} OK?(ac′‘alloc(pm′)(size′)(s′))
-------------------------------------------------------------------------------
 {1}  ac1′ = ac′
 {2}  ac′ = ac2′
 {3}  (ac2′‘memory_pool(pm′)(s′) ⊆ ac1′‘allocated(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
       ∧
       block_is_free(ac2′‘memory_pool(pm′)(s′),
                     at′‘info
                       WITH [(ac′)
                               := cons((data(ac′‘alloc(pm′)(size′)(s′)), size′),
                                       at′‘info(ac′))]
                            (ac1′))
 {4}  data(ac′‘alloc(pm′)(size′)(s′)) = null_address
```

Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_const_private_memory_allocated,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_alloc.1.2.2`.
`allocation_alloc.2:`

```
{-1}  plain_memory?(pm′)
{-2}  allocator?(pm′)(ac′)
{-3}  at′‘hierarchy‘nodes(ac′)
{-4}  size′ > 0
{-5}  pm′‘states(s′)
{-6}  allocation_ok?(pm′)(at′, s′)
{-7}  OK?(ac′‘alloc(pm′)(size′)(s′))
-------------------------------------------------------------------------------
 {1}  ∀ (ac1, ac2, ac3: (at′‘hierarchy‘nodes)):
        at′‘hierarchy‘edges(ac1, ac2) ∧ at′‘hierarchy‘edges(ac1, ac3) ⊃
          ac2 = ac3 ∨
            disjoint?(ac2‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))),
                      ac3‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
 {2}  data(ac′‘alloc(pm′)(size′)(s′)) = null_address
```

Repeatedly Skolemizing and flattening,
Using lemma allocation_subnodes_disjoint,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on ac2!1 = ac!1,

we get 2 subgoals:
`allocation_alloc.2.1:`

```
{-1}  ac2′ = ac′
{-2}  disjoint?(ac2′`memory_pool(pm′)(s′), ac3′`memory_pool(pm′)(s′))
{-3}  allocator?(pm′)(ac1′)
{-4}  at′`hierarchy`nodes(ac1′)
{-5}  allocator?(pm′)(ac2′)
{-6}  at′`hierarchy`nodes(ac2′)
{-7}  allocator?(pm′)(ac3′)
{-8}  at′`hierarchy`nodes(ac3′)
{-9}  at′`hierarchy`edges(ac1′, ac2′)
{-10} at′`hierarchy`edges(ac1′, ac3′)
{-11} plain_memory?(pm′)
{-12} allocator?(pm′)(ac′)
{-13} at′`hierarchy`nodes(ac′)
{-14} size′ > 0
{-15} pm′`states(s′)
{-16} allocation_ok?(pm′)(at′, s′)
{-17} OK?(ac′`alloc(pm′)(size′)(s′))
-------------------------------------------------------------------
 {1}  ac2′ = ac3′
 {2}  disjoint?(ac2′`memory_pool(pm′)(state(ac′`alloc(pm′)(size′)(s′))),
                ac3′`memory_pool(pm′)(state(ac′`alloc(pm′)(size′)(s′))))
 {3}  data(ac′`alloc(pm′)(size′)(s′)) = null_address
```

Replacing using formula -1,
Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_const_private_memory_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_alloc.2.1`.
`allocation_alloc.2.2:`

```
{-1}  disjoint?(ac2′`memory_pool(pm′)(s′), ac3′`memory_pool(pm′)(s′))
{-2}  allocator?(pm′)(ac1′)
{-3}  at′`hierarchy`nodes(ac1′)
{-4}  allocator?(pm′)(ac2′)
{-5}  at′`hierarchy`nodes(ac2′)
{-6}  allocator?(pm′)(ac3′)
{-7}  at′`hierarchy`nodes(ac3′)
{-8}  at′`hierarchy`edges(ac1′, ac2′)
{-9}  at′`hierarchy`edges(ac1′, ac3′)
{-10} plain_memory?(pm′)
{-11} allocator?(pm′)(ac′)
{-12} at′`hierarchy`nodes(ac′)
{-13} size′ > 0
{-14} pm′`states(s′)
{-15} allocation_ok?(pm′)(at′, s′)
{-16} OK?(ac′`alloc(pm′)(size′)(s′))
-------------------------------------------------------------------
 {1}  ac2′ = ac′
 {2}  ac2′ = ac3′
 {3}  disjoint?(ac2′`memory_pool(pm′)(state(ac′`alloc(pm′)(size′)(s′))),
                ac3′`memory_pool(pm′)(state(ac′`alloc(pm′)(size′)(s′))))
 {4}  data(ac′`alloc(pm′)(size′)(s′)) = null_address
```

Case splitting on ac3!1 = ac!1,

we get 2 subgoals:

`allocation_alloc.2.2.1:`

{-1}	$ac3' = ac'$
{-2}	disjoint?(ac2'`memory_pool(pm')(s'), ac3'`memory_pool(pm')(s'))
{-3}	allocator?(pm')(ac1')
{-4}	at'`hierarchy`nodes(ac1')
{-5}	allocator?(pm')(ac2')
{-6}	at'`hierarchy`nodes(ac2')
{-7}	allocator?(pm')(ac3')
{-8}	at'`hierarchy`nodes(ac3')
{-9}	at'`hierarchy`edges(ac1', ac2')
{-10}	at'`hierarchy`edges(ac1', ac3')
{-11}	plain_memory?(pm')
{-12}	allocator?(pm')(ac')
{-13}	at'`hierarchy`nodes(ac')
{-14}	size' > 0
{-15}	pm'`states(s')
{-16}	allocation_ok?(pm')(at', s')
{-17}	OK?(ac'`alloc(pm')(size')(s'))
{1}	$ac2' = ac'$
{2}	$ac2' = ac3'$
{3}	disjoint?(ac2'`memory_pool(pm')(state(ac'`alloc(pm')(size')(s'))), ac3'`memory_pool(pm')(state(ac'`alloc(pm')(size')(s'))))
{4}	data(ac'`alloc(pm')(size')(s')) = null_address

Replacing using formula -1,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.2.2.1`.

`allocation_alloc.2.2.2:`

```
{-1}  disjoint?(ac2′‘memory_pool(pm′)(s′), ac3′‘memory_pool(pm′)(s′))
{-2}  allocator?(pm′)(ac1′)
{-3}  at′‘hierarchy‘nodes(ac1′)
{-4}  allocator?(pm′)(ac2′)
{-5}  at′‘hierarchy‘nodes(ac2′)
{-6}  allocator?(pm′)(ac3′)
{-7}  at′‘hierarchy‘nodes(ac3′)
{-8}  at′‘hierarchy‘edges(ac1′, ac2′)
{-9}  at′‘hierarchy‘edges(ac1′, ac3′)
{-10} plain_memory?(pm′)
{-11} allocator?(pm′)(ac′)
{-12} at′‘hierarchy‘nodes(ac′)
{-13} size′ > 0
{-14} pm′‘states(s′)
{-15} allocation_ok?(pm′)(at′, s′)
{-16} OK?(ac′‘alloc(pm′)(size′)(s′))
--------------------------------------------------------------------------
 {1}  ac3′ = ac′
 {2}  ac2′ = ac′
 {3}  ac2′ = ac3′
 {4}  disjoint?(ac2′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))),
                ac3′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
 {5}  data(ac′‘alloc(pm′)(size′)(s′)) = null_address
```

Using lemma allocation_alloc_other_private_memory,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.2.2.2`.

`allocation_alloc.3:`

```
{-1}  plain_memory?(pm′)
{-2}  allocator?(pm′)(ac′)
{-3}  at′‘hierarchy‘nodes(ac′)
{-4}  size′ > 0
{-5}  pm′‘states(s′)
{-6}  allocation_ok?(pm′)(at′, s′)
{-7}  OK?(ac′‘alloc(pm′)(size′)(s′))
--------------------------------------------------------------------------
 {1}  ∀ (ac1, ac2: (roots(at′‘hierarchy))):
        ac1 = ac2 ∨
         disjoint?(ac1‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))),
                   ac2‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
 {2}  data(ac′‘alloc(pm′)(size′)(s′)) = null_address
```

Repeatedly Skolemizing and flattening,

Using lemma allocation_roots_disjoint,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on ac1!1 = ac!1,

we get 2 subgoals:

`allocation_alloc.3.1:`

{-1}	$ac1' = ac'$
{-2}	disjoint?(ac1'‘memory_pool(pm')(s'), ac2'‘memory_pool(pm')(s'))
{-3}	allocator?(pm')(ac1')
{-4}	at'‘hierarchy‘nodes(ac1')
{-5}	roots(at'‘hierarchy)(ac1')
{-6}	allocator?(pm')(ac2')
{-7}	at'‘hierarchy‘nodes(ac2')
{-8}	roots(at'‘hierarchy)(ac2')
{-9}	plain_memory?(pm')
{-10}	allocator?(pm')(ac')
{-11}	at'‘hierarchy‘nodes(ac')
{-12}	$size' > 0$
{-13}	pm'‘states(s')
{-14}	allocation_ok?(pm')(at', s')
{-15}	OK?(ac'‘alloc(pm')(size')(s'))
{1}	$ac1' = ac2'$
{2}	disjoint?(ac1'‘memory_pool(pm')(state(ac'‘alloc(pm')(size')(s'))), ac2'‘memory_pool(pm')(state(ac'‘alloc(pm')(size')(s'))))
{3}	data(ac'‘alloc(pm')(size')(s')) = null_address

Replacing using formula -1,

Using lemma allocation_alloc_other_private_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.3.1`.

`allocation_alloc.3.2:`

{-1}	disjoint?(ac1'‘memory_pool(pm')(s'), ac2'‘memory_pool(pm')(s'))
{-2}	allocator?(pm')(ac1')
{-3}	at'‘hierarchy‘nodes(ac1')
{-4}	roots(at'‘hierarchy)(ac1')
{-5}	allocator?(pm')(ac2')
{-6}	at'‘hierarchy‘nodes(ac2')
{-7}	roots(at'‘hierarchy)(ac2')
{-8}	plain_memory?(pm')
{-9}	allocator?(pm')(ac')
{-10}	at'‘hierarchy‘nodes(ac')
{-11}	$size' > 0$
{-12}	pm'‘states(s')
{-13}	allocation_ok?(pm')(at', s')
{-14}	OK?(ac'‘alloc(pm')(size')(s'))
{1}	$ac1' = ac'$
{2}	$ac1' = ac2'$
{3}	disjoint?(ac1'‘memory_pool(pm')(state(ac'‘alloc(pm')(size')(s'))), ac2'‘memory_pool(pm')(state(ac'‘alloc(pm')(size')(s'))))
{4}	data(ac'‘alloc(pm')(size')(s')) = null_address

Case splitting on ac2!1 = ac!1,

we get 2 subgoals:

`allocation_alloc.3.2.1:`

{-1}	ac2′ = ac′
{-2}	disjoint?(ac1′‘memory_pool(pm′)(s′), ac2′‘memory_pool(pm′)(s′))
{-3}	allocator?(pm′)(ac1′)
{-4}	at′‘hierarchy‘nodes(ac1′)
{-5}	roots(at′‘hierarchy)(ac1′)
{-6}	allocator?(pm′)(ac2′)
{-7}	at′‘hierarchy‘nodes(ac2′)
{-8}	roots(at′‘hierarchy)(ac2′)
{-9}	plain_memory?(pm′)
{-10}	allocator?(pm′)(ac′)
{-11}	at′‘hierarchy‘nodes(ac′)
{-12}	size′ > 0
{-13}	pm′‘states(s′)
{-14}	allocation_ok?(pm′)(at′, s′)
{-15}	OK?(ac′‘alloc(pm′)(size′)(s′))
{1}	ac1′ = ac′
{2}	ac1′ = ac2′
{3}	disjoint?(ac1′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))), ac2′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
{4}	data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Replacing using formula -1,
Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_const_private_memory_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_alloc.3.2.1`.
`allocation_alloc.3.2.2:`

{-1}	disjoint?(ac1′‘memory_pool(pm′)(s′), ac2′‘memory_pool(pm′)(s′))
{-2}	allocator?(pm′)(ac1′)
{-3}	at′‘hierarchy‘nodes(ac1′)
{-4}	roots(at′‘hierarchy)(ac1′)
{-5}	allocator?(pm′)(ac2′)
{-6}	at′‘hierarchy‘nodes(ac2′)
{-7}	roots(at′‘hierarchy)(ac2′)
{-8}	plain_memory?(pm′)
{-9}	allocator?(pm′)(ac′)
{-10}	at′‘hierarchy‘nodes(ac′)
{-11}	size′ > 0
{-12}	pm′‘states(s′)
{-13}	allocation_ok?(pm′)(at′, s′)
{-14}	OK?(ac′‘alloc(pm′)(size′)(s′))
{1}	ac2′ = ac′
{2}	ac1′ = ac′
{3}	ac1′ = ac2′
{4}	disjoint?(ac1′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))), ac2′‘memory_pool(pm′)(state(ac′‘alloc(pm′)(size′)(s′))))
{5}	data(ac′‘alloc(pm′)(size′)(s′)) = null_address

Using lemma allocation_alloc_other_private_memory,
Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_memory_pool,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.3.2.2`.

`allocation_alloc.4`:

```
{-1}  plain_memory?(pm')
{-2}  allocator?(pm')(ac')
{-3}  at'`hierarchy`nodes(ac')
{-4}  size' > 0
{-5}  pm'`states(s')
{-6}  allocation_ok?(pm')(at', s')
{-7}  OK?(ac'`alloc(pm')(size')(s'))
------------------------------------------------------------------------
{1}   ∀ (ac1, ac2: (at'`hierarchy`nodes)):
        ac1 = ac2 ∨ disjoint?(ac1`private_mem(pm'), ac2`private_mem(pm'))
{2}   data(ac'`alloc(pm')(size')(s')) = null_address
```

Repeatedly Skolemizing and flattening,

Using lemma allocation_private_memory_disjoint,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocation_alloc.4`.

`allocation_alloc.5`:

```
{-1}  plain_memory?(pm')
{-2}  allocator?(pm')(ac')
{-3}  at'`hierarchy`nodes(ac')
{-4}  size' > 0
{-5}  pm'`states(s')
{-6}  allocation_ok?(pm')(at', s')
{-7}  OK?(ac'`alloc(pm')(size')(s'))
------------------------------------------------------------------------
{1}   ∀ (ac: (at'`hierarchy`nodes)):
        consistent_allocator?(pm')
                             (ac,
                                at'`info
                                   WITH [(ac')
                                           := cons((data(ac'`alloc(pm')(size')(s')),
                                                    size'),
                                                   at'`info(ac'))]
                                      (ac))
                             (state(ac'`alloc(pm')(size')(s')))
{2}   data(ac'`alloc(pm')(size')(s')) = null_address
```

Repeatedly Skolemizing and flattening,

Using lemma allocation_consistent_allocator,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on ac!1 = ac!2,

we get 2 subgoals:

`allocation_alloc.5.1:`

```
  {-1}   ac′ = ac″
  {-2}   consistent_allocator?(pm′)(ac″, at′`info(ac″))(s′)
  {-3}   allocator?(pm′)(ac″)
  {-4}   at′`hierarchy`nodes(ac″)
  {-5}   plain_memory?(pm′)
  {-6}   allocator?(pm′)(ac′)
  {-7}   at′`hierarchy`nodes(ac′)
  {-8}   size′ > 0
  {-9}   pm′`states(s′)
  {-10}  allocation_ok?(pm′)(at′, s′)
  {-11}  OK?(ac′`alloc(pm′)(size′)(s′))
  ------------------------------------------------------------------------
  {1}    consistent_allocator?(pm′)
                               (ac″,
                                  at′`info
                                    WITH [(ac′)
                                            := cons((data(ac′`alloc(pm′)(size′)(s′)), size′),
                                                    at′`info(ac′))]
                                       (ac″))
                               (state(ac′`alloc(pm′)(size′)(s′)))
  {2}    data(ac′`alloc(pm′)(size′)(s′)) = null_address
```

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using consistent_allocator_alloc, matching in *,
This completes the proof of `allocation_alloc.5.1`.
`allocation_alloc.5.2:`

```
  {-1}   consistent_allocator?(pm′)(ac″, at′`info(ac″))(s′)
  {-2}   allocator?(pm′)(ac″)
  {-3}   at′`hierarchy`nodes(ac″)
  {-4}   plain_memory?(pm′)
  {-5}   allocator?(pm′)(ac′)
  {-6}   at′`hierarchy`nodes(ac′)
  {-7}   size′ > 0
  {-8}   pm′`states(s′)
  {-9}   allocation_ok?(pm′)(at′, s′)
  {-10}  OK?(ac′`alloc(pm′)(size′)(s′))
  ------------------------------------------------------------------------
  {1}    ac′ = ac″
  {2}    consistent_allocator?(pm′)
                               (ac″,
                                  at′`info
                                    WITH [(ac′)
                                            := cons((data(ac′`alloc(pm′)(size′)(s′)), size′),
                                                    at′`info(ac′))]
                                       (ac″))
                               (state(ac′`alloc(pm′)(size′)(s′)))
  {3}    data(ac′`alloc(pm′)(size′)(s′)) = null_address
```

Simplifying, rewriting, and recording with decision procedures,
Using lemma consistent_allocator_state_change,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocation_alloc_other_private_memory,
Simplifying, rewriting, and recording with decision procedures,

Using lemma allocator_const_private_memory_allocated,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocation_alloc.5.2`.
Q.E.D.

C.10 Proofs for Allocator (allocators.pvs)

C.10.1 Allocator.null_address_TCC1

Terse proof for `null_address_TCC1`.

`null_address_TCC1:`

{1} Mem?(Mem(0)‘type_of)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `null_address_TCC1`.
Q.E.D.

C.10.2 Allocator.allocator?_TCC1

Terse proof for `allocator?_TCC1`.

`allocator?_TCC1:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        (∀ (s: State):
           (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s) ⊃
            (∀ (size: posnat):
               disjoint?(ac‘allocated(pm)(s),
                         address_block(data(ac‘alloc(pm)(size)(s)), size))
                ∧
                OK?(ac‘alloc(pm)(size)(s)) ∧
                 ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                ⊃
                OK?[State, Address](ac‘alloc(pm)(size)(s)) ∨
                 Exception?[State, Address](ac‘alloc(pm)(size)(s))))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `allocator?_TCC1`.
Q.E.D.

C.10.3 Allocator.allocator?_TCC2

Terse proof for `allocator?_TCC2`.

`allocator?_TCC2:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        (∀ (s: State):
           (∀ (addr: Address):
              OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
            ∧
            (∀ (size: posnat):
               OK?(ac‘alloc(pm)(size)(s)) ∧
                ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                ⊃
                disjoint?(ac‘allocated(pm)(s),
                          address_block(data(ac‘alloc(pm)(size)(s)), size))
                 ∧
                 ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
                  (ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))
                  ∧
                  ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
                   ac‘memory_pool(pm)(s))
             ∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
            ⊃
            (∀ (addr1: Address):
               OK?(ac‘free(pm)(addr1)(s)) ⊃
                OK?[State, Unit](ac‘free(pm)(addr1)(s)) ∨
                 Exception?[State, Unit](ac‘free(pm)(addr1)(s))))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `allocator?_TCC2`.

Q.E.D.

C.10.4 Allocator.allocator?_TCC3

Terse proof for `allocator?_TCC3`.

`allocator?_TCC3:`

```
{1}   ∀ (pm: (plain_memory?[State]), ac: Allocator):
        (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
         (∀ (s: State):
              (∀ (addr: Address):
                   OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
               ∧
               (∀ (size: posnat):
                    OK?(ac‘alloc(pm)(size)(s)) ∧
                     ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                     ⊃
                     disjoint?(ac‘allocated(pm)(s),
                                  address_block(data(ac‘alloc(pm)(size)(s)), size))
                      ∧
                      ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
                       (ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))
                       ∧
                       ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
                        ac‘memory_pool(pm)(s))
                ∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
               ⊃
               (∀ (addr1: Address):
                    OK?(ac‘free(pm)(addr1)(s)) ⊃
                     up?[posnat](ac‘freed_size(pm)(addr1)(s))))
```

Repeatedly Skolemizing and flattening,

Rewriting using -3, matching in *,

This completes the proof of `allocator?_TCC3`.

Q.E.D.

C.10.5 Allocator.allocator?_TCC4

Terse proof for `allocator?_TCC4`.

`allocator?_TCC4:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        (∀ (s: State):
           (∀ (s_2: State):
              pm‘states(s_2) ∧ stays_unchanged(pm)(s, s_2, ac‘private_mem(pm)) ⊃
               ac‘memory_pool(pm)(s) = ac‘memory_pool(pm)(s_2) ∧
                ac‘allocated(pm)(s) = ac‘allocated(pm)(s_2))
         ∧
         (∀ (addr: Address):
              OK?(ac‘free(pm)(addr)(s)) ⊃
               ac‘allocated(pm)(state(ac‘free(pm)(addr)(s))) =
                (ac‘allocated(pm)(s) \ address_block(addr, down(ac‘freed_size(pm)(addr)(s))))
                ∧
                ac‘memory_pool(pm)(state(ac‘free(pm)(addr)(s))) =
                 ac‘memory_pool(pm)(s))
          ∧
          (∀ (addr: Address):
               OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
           ∧
           (∀ (size: posnat):
                OK?(ac‘alloc(pm)(size)(s)) ∧
                 ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                 ⊃
                 disjoint?(ac‘allocated(pm)(s),
                                address_block(data(ac‘alloc(pm)(size)(s)), size))
                  ∧
                  ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
                   (ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))
                   ∧
                   ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
                    ac‘memory_pool(pm)(s))
             ∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
         ⊃
         (∀ (size1: posnat):
              has_next_state(ac‘alloc(pm)(size1)(s)) ⊃
               OK?[State, Address](ac‘alloc(pm)(size1)(s)) ∨
                Exception?[State, Address](ac‘alloc(pm)(size1)(s))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of has_next_state,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `allocator?_TCC4`.

Q.E.D.

C.10.6 Allocator.allocator?_TCC5

Terse proof for `allocator?_TCC5`.

`allocator?_TCC5:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        (∀ (s: State):
           (∀ (size: posnat):
              has_next_state(ac‘alloc(pm)(size)(s)) ⊃
               pm‘states(state(ac‘alloc(pm)(size)(s))))
            ∧
            (∀ (s_2: State):
               pm‘states(s_2) ∧ stays_unchanged(pm)(s, s_2, ac‘private_mem(pm)) ⊃
                ac‘memory_pool(pm)(s) = ac‘memory_pool(pm)(s_2) ∧
                 ac‘allocated(pm)(s) = ac‘allocated(pm)(s_2))
             ∧
             (∀ (addr: Address):
                OK?(ac‘free(pm)(addr)(s)) ⊃
                 ac‘allocated(pm)(state(ac‘free(pm)(addr)(s))) =
                  (ac‘allocated(pm)(s) \ address_block(addr, down(ac‘freed_size(pm)(addr)(s))))
                  ∧
                  ac‘memory_pool(pm)(state(ac‘free(pm)(addr)(s))) =
                   ac‘memory_pool(pm)(s))
              ∧
              (∀ (addr: Address):
                 OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
               ∧
               (∀ (size: posnat):
                  OK?(ac‘alloc(pm)(size)(s)) ∧
                   ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                   ⊃
                   disjoint?(ac‘allocated(pm)(s),
                             address_block(data(ac‘alloc(pm)(size)(s)), size))
                    ∧
                    ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
                     (ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))
                     ∧
                     ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
                      ac‘memory_pool(pm)(s))
                ∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
            ⊃
            (∀ (addr1: Address):
               has_next_state(ac‘free(pm)(addr1)(s)) ⊃
                OK?[State, Unit](ac‘free(pm)(addr1)(s)) ∨
                 Exception?[State, Unit](ac‘free(pm)(addr1)(s))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of has_next_state,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `allocator?_TCC5`.

Q.E.D.

C.10.7 Allocator.allocator?_TCC6

Terse proof for `allocator?_TCC6`.

`allocator?_TCC6:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        (∀ (s: State):
           (∀ (addr: Address):
              has_next_state(ac‘free(pm)(addr)(s)) ⊃
               pm‘states(state(ac‘free(pm)(addr)(s))))
            ∧
            (∀ (size: posnat):
               has_next_state(ac‘alloc(pm)(size)(s)) ⊃
                pm‘states(state(ac‘alloc(pm)(size)(s))))
             ∧
             (∀ (s_2: State):
                pm‘states(s_2) ∧ stays_unchanged(pm)(s, s_2, ac‘private_mem(pm)) ⊃
                 ac‘memory_pool(pm)(s) = ac‘memory_pool(pm)(s_2) ∧
                  ac‘allocated(pm)(s) = ac‘allocated(pm)(s_2))
              ∧
              (∀ (addr: Address):
                 OK?(ac‘free(pm)(addr)(s)) ⊃
                  ac‘allocated(pm)(state(ac‘free(pm)(addr)(s))) =
                   (ac‘allocated(pm)(s) \ address_block(addr, down(ac‘freed_size(pm)(addr)(s))))
                   ∧
                   ac‘memory_pool(pm)(state(ac‘free(pm)(addr)(s))) =
                    ac‘memory_pool(pm)(s))
               ∧
               (∀ (addr: Address):
                  OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
                ∧
                (∀ (size: posnat):
                   OK?(ac‘alloc(pm)(size)(s)) ∧
                    ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                    ⊃
                    disjoint?(ac‘allocated(pm)(s),
                              address_block(data(ac‘alloc(pm)(size)(s)), size))
                     ∧
                     ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
                      (ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))
                      ∧
                      ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
                       ac‘memory_pool(pm)(s))
                 ∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
            ⊃
            (∀ (size1: posnat):
               OK?(ac‘alloc(pm)(size1)(s)) ∧
                ¬ data(ac‘alloc(pm)(size1)(s)) = null_address
                ⊃
                OK?[State, Address](ac‘alloc(pm)(size1)(s)) ∨
                 Exception?[State, Address](ac‘alloc(pm)(size1)(s))))
```

Repeatedly Skolemizing and flattening,

This completes the proof of `allocator?_TCC6`.

Q.E.D.

C.10.8 Allocator.allocator?_TCC7

Terse proof for `allocator?_TCC7`.

`allocator?_TCC7:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: Allocator):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        (∀ (s: State):
           (∀ (addr: Address):
              has_next_state(ac‘free(pm)(addr)(s)) ⊃
               pm‘states(state(ac‘free(pm)(addr)(s))))
            ∧
            (∀ (size: posnat):
               has_next_state(ac‘alloc(pm)(size)(s)) ⊃
                pm‘states(state(ac‘alloc(pm)(size)(s))))
             ∧
             (∀ (s_2: State):
                pm‘states(s_2) ∧ stays_unchanged(pm)(s, s_2, ac‘private_mem(pm)) ⊃
                 ac‘memory_pool(pm)(s) = ac‘memory_pool(pm)(s_2) ∧
                  ac‘allocated(pm)(s) = ac‘allocated(pm)(s_2))
              ∧
              (∀ (addr: Address):
                 OK?(ac‘free(pm)(addr)(s)) ⊃
                  ac‘allocated(pm)(state(ac‘free(pm)(addr)(s))) =
                   (ac‘allocated(pm)(s) \ address_block(addr,  down(ac‘freed_size(pm)(addr)(s))))
                   ∧
                   ac‘memory_pool(pm)(state(ac‘free(pm)(addr)(s))) =
                    ac‘memory_pool(pm)(s))
               ∧
               (∀ (addr: Address):
                  OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
                ∧
                (∀ (size: posnat):
                   OK?(ac‘alloc(pm)(size)(s)) ∧
                    ¬ data(ac‘alloc(pm)(size)(s)) = null_address
                    ⊃
                    disjoint?(ac‘allocated(pm)(s),
                              address_block(data(ac‘alloc(pm)(size)(s)), size))
                     ∧
                     ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
                      (ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)),  size))
                      ∧
                      ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
                       ac‘memory_pool(pm)(s))
                 ∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
            ⊃
            (∀ (size1: posnat):
               OK?(ac‘alloc(pm)(size1)(s)) ∧
                ¬ data(ac‘alloc(pm)(size1)(s)) = null_address
                ⊃ pm‘states(state[State, Address](ac‘alloc(pm)(size1)(s)))))
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of has_next_state,
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `allocator?_TCC7`.

Q.E.D.

C.10.9 Allocator.allocator?_TCC8

Terse proof for `allocator?_TCC8`.

allocator?_TCC8:

{1} ∀ (pm: (plain_memory?[State]), ac: Allocator):
(ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
(∀ (s: State):
(∀ (size: posnat):
OK?(ac‘alloc(pm)(size)(s)) ∧
¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
only_changes(pm)(s, state(ac‘alloc(pm)(size)(s)), ac‘private_mem(pm)))
∧
(∀ (addr: Address):
has_next_state(ac‘free(pm)(addr)(s)) ⊃
pm‘states(state(ac‘free(pm)(addr)(s))))
∧
(∀ (size: posnat):
has_next_state(ac‘alloc(pm)(size)(s)) ⊃
pm‘states(state(ac‘alloc(pm)(size)(s))))
∧
(∀ (s_2: State):
pm‘states(s_2) ∧ stays_unchanged(pm)(s, s_2, ac‘private_mem(pm)) ⊃
ac‘memory_pool(pm)(s) = ac‘memory_pool(pm)(s_2) ∧
ac‘allocated(pm)(s) = ac‘allocated(pm)(s_2))
∧
(∀ (addr: Address):
OK?(ac‘free(pm)(addr)(s)) ⊃
ac‘allocated(pm)(state(ac‘free(pm)(addr)(s))) =
(ac‘allocated(pm)(s) \ address_block(addr, down(ac‘freed_size(pm)(addr)(s))))
∧
ac‘memory_pool(pm)(state(ac‘free(pm)(addr)(s))) =
ac‘memory_pool(pm)(s))
∧
(∀ (addr: Address):
OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))
∧
(∀ (size: posnat):
OK?(ac‘alloc(pm)(size)(s)) ∧
¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
disjoint?(ac‘allocated(pm)(s),
address_block(data(ac‘alloc(pm)(size)(s)), size))
∧
ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
(ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))
∧
ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) =
ac‘memory_pool(pm)(s))
∧ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s)) ∧ pm‘states(s)
⊃
(∀ (free_addr: Address):
OK?(ac‘free(pm)(free_addr)(s)) ⊃
pm‘states(state[State, Unit](ac‘free(pm)(free_addr)(s)))))

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of has_next_state,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator?_TCC8`.
Q.E.D.

C.10.10 Allocator.allocator_subset_private_memory_plain_memory

Terse proof for `allocator_subset_private_memory_plain_memory`.
`allocator_subset_private_memory_plain_memory:`

```
{1}  ∀ (pm: (plain_memory?), ac: (allocator?(pm))):
       (ac‘private_mem(pm) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_subset_private_memory_plain_memory`.
Q.E.D.

C.10.11 Allocator.allocator_private_memory_in_plain_memory

Terse proof for `allocator_private_memory_in_plain_memory`.
`allocator_private_memory_in_plain_memory:`

```
{1}  ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address):
       ac‘private_mem(pm)(addr) ⊃ union(pm‘ro_addr, pm‘rw_addr)(addr)
```

Repeatedly Skolemizing and flattening,
Using lemma allocator_subset_private_memory_plain_memory,
Using lemma subset_member,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_private_memory_in_plain_memory`.
Q.E.D.

C.10.12 Allocator.allocator_subset_allocated_memory_pool

Terse proof for `allocator_subset_allocated_memory_pool`.
`allocator_subset_allocated_memory_pool:`

```
{1}  ∀ (pm: (plain_memory?), ac: (allocator?(pm)), s: State):
       pm‘states(s) ⊃ (ac‘allocated(pm)(s) ⊆ ac‘memory_pool(pm)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_subset_allocated_memory_pool`.
Q.E.D.

C.10.13 Allocator.allocator_alloc_disjoint

Terse proof for `allocator_alloc_disjoint`.

`allocator_alloc_disjoint:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
disjoint?(ac‘allocated(pm)(s),
address_block(data(ac‘alloc(pm)(size)(s)), size))

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s',
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_alloc_disjoint`.
Q.E.D.

C.10.14 Allocator.allocator_allocated_alloc_TCC1

Terse proof for `allocator_allocated_alloc_TCC1`.

`allocator_allocated_alloc_TCC1:`

{1} ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
OK?[State, Address](ac‘alloc(pm)(size)(s)) ∨
Exception?[State, Address](ac‘alloc(pm)(size)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `allocator_allocated_alloc_TCC1`.
Q.E.D.

C.10.15 Allocator.allocator_allocated_alloc

Terse proof for `allocator_allocated_alloc`.

`allocator_allocated_alloc:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s))) =
(ac‘allocated(pm)(s) ∪ address_block(data(ac‘alloc(pm)(size)(s)), size))

Repeatedly Skolemizing and flattening,

Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s',
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_allocated_alloc`.
Q.E.D.

C.10.16 Allocator.allocator_memory_pool_alloc

Terse proof for `allocator_memory_pool_alloc`.
`allocator_memory_pool_alloc:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃ ac‘memory_pool(pm)(state(ac‘alloc(pm)(size)(s))) = ac‘memory_pool(pm)(s)

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s',
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_memory_pool_alloc`.
Q.E.D.

C.10.17 Allocator.allocator_subset_address_block_alloc_allocated

Terse proof for `allocator_subset_address_block_alloc_allocated`.
`allocator_subset_address_block_alloc_allocated:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, addr: Address, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
(address_block(data(ac‘alloc(pm)(size)(s)), size) ⊆ ac‘allocated(pm)(state(ac‘alloc(pm)(size)(s)

Repeatedly Skolemizing and flattening,
Using lemma allocator_allocated_alloc,
Simplifying, rewriting, and recording with decision procedures,
Applying union_subset3 where a gets address_block(data(ac!1‘alloc(pm!1)(size!1)(s!1)), size!1), b gets ac!1‘allocated(pm!1)(s!1),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_subset_address_block_alloc_allocated`.
Q.E.D.

C.10.18 Allocator.allocator_free_freed_size

Terse proof for `allocator_free_freed_size`.

`allocator_free_freed_size:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
pm‘states(s) ⊃
(OK?(ac‘free(pm)(addr)(s)) ≡ up?(ac‘freed_size(pm)(addr)(s)))

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `allocator_free_freed_size`.
Q.E.D.

C.10.19 Allocator.allocator_allocated_free_TCC1

Terse proof for `allocator_allocated_free_TCC1`.

`allocator_allocated_free_TCC1:`

{1} ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, s: State):
pm‘states(s) ∧ OK?(ac‘free(pm)(addr)(s)) ⊃
OK?[State, Unit](ac‘free(pm)(addr)(s)) ∨
Exception?[State, Unit](ac‘free(pm)(addr)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `allocator_allocated_free_TCC1`.
Q.E.D.

C.10.20 Allocator.allocator_allocated_free_TCC2

Terse proof for `allocator_allocated_free_TCC2`.

`allocator_allocated_free_TCC2:`

{1} ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, s: State):
pm‘states(s) ∧ OK?(ac‘free(pm)(addr)(s)) ⊃
up?[posnat](ac‘freed_size(pm)(addr)(s))

Repeatedly Skolemizing and flattening,
Rewriting using allocator_free_freed_size, matching in *,
This completes the proof of `allocator_allocated_free_TCC2`.
Q.E.D.

C.10.21 Allocator.allocator_allocated_free

Terse proof for `allocator_allocated_free`.

`allocator_allocated_free:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
pm‘states(s) ∧ OK?(ac‘free(pm)(addr)(s)) ⊃
ac‘allocated(pm)(state(ac‘free(pm)(addr)(s))) =
(ac‘allocated(pm)(s) \ address_block(addr, down(ac‘freed_size(pm)(addr)(s))))

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s',
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_allocated_free`.
Q.E.D.

C.10.22 Allocator.allocator_memory_pool_free

Terse proof for `allocator_memory_pool_free`.
`allocator_memory_pool_free:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
pm‘states(s) ∧ OK?(ac‘free(pm)(addr)(s)) ⊃
ac‘memory_pool(pm)(state(ac‘free(pm)(addr)(s))) = ac‘memory_pool(pm)(s)

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s',
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_memory_pool_free`.
Q.E.D.

C.10.23 Allocator.allocator_const_private_memory_memory_pool

Terse proof for `allocator_const_private_memory_memory_pool`.
`allocator_const_private_memory_memory_pool:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), s_1, s_2: State):
pm‘states(s_1) ∧ pm‘states(s_2) ∧ stays_unchanged(pm)(s_1, s_2, ac‘private_mem(pm)) ⊃
ac‘memory_pool(pm)(s_1) = ac‘memory_pool(pm)(s_2)

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s_1',
Simplifying, rewriting, and recording with decision procedures,

Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_const_private_memory_memory_pool`.
Q.E.D.

C.10.24 Allocator.allocator_const_private_memory_allocated

Terse proof for `allocator_const_private_memory_allocated`.
`allocator_const_private_memory_allocated`:

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), s_1, s_2: State):
pm‘states(s_1) ∧ pm‘states(s_2) ∧ stays_unchanged(pm)(s_1, s_2, ac‘private_mem(pm)) ⊃
ac‘allocated(pm)(s_1) = ac‘allocated(pm)(s_2)

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -3 with the terms: s_1',
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -7 with the terms: s_2',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_const_private_memory_allocated`.
Q.E.D.

C.10.25 Allocator.allocator_next_state_alloc_TCC1

Terse proof for `allocator_next_state_alloc_TCC1`.
`allocator_next_state_alloc_TCC1`:

{1} ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧ has_next_state(ac‘alloc(pm)(size)(s)) ⊃
OK?[State, Address](ac‘alloc(pm)(size)(s)) ∨
Exception?[State, Address](ac‘alloc(pm)(size)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `allocator_next_state_alloc_TCC1`.
Q.E.D.

C.10.26 Allocator.allocator_next_state_alloc

Terse proof for `allocator_next_state_alloc`.
`allocator_next_state_alloc`:

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧ has_next_state(ac‘alloc(pm)(size)(s)) ⊃
pm‘states(state(ac‘alloc(pm)(size)(s)))

Repeatedly Skolemizing and flattening,

Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_next_state_alloc`.
Q.E.D.

C.10.27 Allocator.allocator_next_state_free_TCC1

Terse proof for `allocator_next_state_free_TCC1`.

`allocator_next_state_free_TCC1`:

```
{1}  ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, s: State):
       pm‘states(s) ∧ has_next_state(ac‘free(pm)(addr)(s)) ⊃
         OK?[State, Unit](ac‘free(pm)(addr)(s)) ∨
           Exception?[State, Unit](ac‘free(pm)(addr)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `allocator_next_state_free_TCC1`.
Q.E.D.

C.10.28 Allocator.allocator_next_state_free

Terse proof for `allocator_next_state_free`.

`allocator_next_state_free`:

```
{1}  ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
       pm‘states(s) ∧ has_next_state(ac‘free(pm)(addr)(s)) ⊃
         pm‘states(state(ac‘free(pm)(addr)(s)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_next_state_free`.
Q.E.D.

C.10.29 Allocator.allocator_only_changes_alloc_TCC1

Terse proof for `allocator_only_changes_alloc_TCC1`.

allocator_only_changes_alloc_TCC1:

{1} ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃ pm‘states(state[State, Address](ac‘alloc(pm)(size)(s)))

Repeatedly Skolemizing and flattening,
Rewriting using allocator_next_state_alloc, matching in *,
Expanding the definition of has_next_state,
which is trivially true.
This completes the proof of allocator_only_changes_alloc_TCC1.
Q.E.D.

C.10.30 Allocator.allocator_only_changes_alloc

Terse proof for allocator_only_changes_alloc.
allocator_only_changes_alloc:

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃ only_changes(pm)(s, state(ac‘alloc(pm)(size)(s)), ac‘private_mem(pm))

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of allocator_only_changes_alloc.
Q.E.D.

C.10.31 Allocator.allocator_only_changes_free_TCC1

Terse proof for allocator_only_changes_free_TCC1.
allocator_only_changes_free_TCC1:

{1} ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), free_addr: Address, s: State):
pm‘states(s) ∧ OK?(ac‘free(pm)(free_addr)(s)) ⊃
pm‘states(state[State, Unit](ac‘free(pm)(free_addr)(s)))

Repeatedly Skolemizing and flattening,
Rewriting using allocator_next_state_free, matching in *,
Expanding the definition of has_next_state,
which is trivially true.
This completes the proof of allocator_only_changes_free_TCC1.
Q.E.D.

C.10.32 Allocator.allocator_only_changes_free

Terse proof for `allocator_only_changes_free`.

`allocator_only_changes_free:`

```
{1}  ∀ (pm: (plain_memory?), ac: (allocator?(pm)), free_addr: Address, s: State):
       pm‘states(s) ∧ OK?(ac‘free(pm)(free_addr)(s)) ⊃
         only_changes(pm)(s, state(ac‘free(pm)(free_addr)(s)), ac‘private_mem(pm))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of allocator?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_only_changes_free`.
Q.E.D.

C.10.33 Allocator.allocator_allocated_memory_read_TCC1

Terse proof for `allocator_allocated_memory_read_TCC1`.

`allocator_allocated_memory_read_TCC1:`

```
{1}  ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, s: State):
       pm‘states(s) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr) ⊃
         OK?[State, Byte](memory_read(pm‘mem)(addr)(s)) ∨
           Exception?[State, Byte](memory_read(pm‘mem)(addr)(s))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of `allocator_allocated_memory_read_TCC1`.
Q.E.D.

C.10.34 Allocator.allocator_allocated_memory_read

Terse proof for `allocator_allocated_memory_read`.

`allocator_allocated_memory_read:`

```
{1}  ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
       pm‘states(s) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr) ⊃
         ac‘allocated(pm)(state(memory_read(pm‘mem)(addr)(s))) = ac‘allocated(pm)(s)
```

Installing automatic rewrites from: plain_memory_states_memory_read
Repeatedly Skolemizing and flattening,
Rewriting using allocator_const_private_memory_allocated, matching in *,
we get 2 subgoals:

`allocator_allocated_memory_read.1:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac′)
{-3} pm′`states(s′)
{-4} union(pm′`ro_addr, pm′`rw_addr)(addr′)

{1} pm′`states(state(memory_read(pm′`mem)(addr′)(s′)))
{2} ac′`allocated(pm′)(state(memory_read(pm′`mem)(addr′)(s′))) =
ac′`allocated(pm′)(s′)

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_allocated_memory_read.1`.
`allocator_allocated_memory_read.2:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac′)
{-3} pm′`states(s′)
{-4} union(pm′`ro_addr, pm′`rw_addr)(addr′)

{1} stays_unchanged(pm′)
(state(memory_read(pm′`mem)(addr′)(s′)), s′, ac′`private_mem(pm′))
{2} ac′`allocated(pm′)(state(memory_read(pm′`mem)(addr′)(s′))) =
ac′`allocated(pm′)(s′)

Rewriting using stays_unchanged_symmetric, matching in *,
Using lemma plain_memory_stays_unchanged_memory_read,
Simplifying, rewriting, and recording with decision procedures,
Using lemma stays_unchanged_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_subset_private_memory_plain_memory, matching in *,
This completes the proof of `allocator_allocated_memory_read.2`.
Q.E.D.

C.10.35 Allocator.allocator_memory_pool_memory_read

Terse proof for `allocator_memory_pool_memory_read`.
`allocator_memory_pool_memory_read:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, s: State):
pm`states(s) ∧ union(pm`ro_addr, pm`rw_addr)(addr) ⊃
ac`memory_pool(pm)(state(memory_read(pm`mem)(addr)(s))) =
ac`memory_pool(pm)(s)

Installing automatic rewrites from: plain_memory_states_memory_read
Repeatedly Skolemizing and flattening,
Rewriting using allocator_const_private_memory_memory_pool, matching in *,
we get 2 subgoals:
`allocator_memory_pool_memory_read.1:`

{-1} plain_memory?(pm′)
{-2} allocator?(pm′)(ac′)
{-3} pm′`states(s′)
{-4} union(pm′`ro_addr, pm′`rw_addr)(addr′)

{1} pm′`states(state(memory_read(pm′`mem)(addr′)(s′)))
{2} ac′`memory_pool(pm′)(state(memory_read(pm′`mem)(addr′)(s′))) =
ac′`memory_pool(pm′)(s′)

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `allocator_memory_pool_memory_read.1`.
`allocator_memory_pool_memory_read.2`:

```
{-1}  plain_memory?(pm')
{-2}  allocator?(pm')(ac')
{-3}  pm'`states(s')
{-4}  union(pm'`ro_addr, pm'`rw_addr)(addr')
------------------------------------------------------------------------
{1}   stays_unchanged(pm')
                    (state(memory_read(pm'`mem)(addr')(s')), s', ac'`private_mem(pm'))
{2}   ac'`memory_pool(pm')(state(memory_read(pm'`mem)(addr')(s'))) =
        ac'`memory_pool(pm')(s')
```

Rewriting using stays_unchanged_symmetric, matching in *,
Using lemma plain_memory_stays_unchanged_memory_read,
Simplifying, rewriting, and recording with decision procedures,
Using lemma stays_unchanged_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_subset_private_memory_plain_memory, matching in *,
This completes the proof of `allocator_memory_pool_memory_read.2`.
Q.E.D.

C.10.36 Allocator.allocator_stays_unchanged_memory_write_private_mem_TCC1

Terse proof for `allocator_stays_unchanged_memory_write_private_mem_TCC1`.
`allocator_stays_unchanged_memory_write_private_mem_TCC1`:

```
{1}   ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, byte: Byte,
           s: State):
        pm`states(s) ∧ pm`rw_addr(addr) ∧ ¬ ac`private_mem(pm)(addr) ⊃
          OK?[State, Unit](memory_write(pm`mem)(addr, byte)(s)) ∨
            Exception?[State, Unit](memory_write(pm`mem)(addr, byte)(s))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_write_ok, matching in *,
This completes the proof of `allocator_stays_unchanged_memory_write_private_mem_TCC1`.
Q.E.D.

C.10.37 Allocator.allocator_stays_unchanged_memory_write_private_mem_TCC2

Terse proof for `allocator_stays_unchanged_memory_write_private_mem_TCC2`.
`allocator_stays_unchanged_memory_write_private_mem_TCC2`:

```
{1}   ∀ (pm: (plain_memory?[State]), ac: (allocator?(pm)), addr: Address, byte: Byte,
           s: State):
        pm`states(s) ∧ pm`rw_addr(addr) ∧ ¬ ac`private_mem(pm)(addr) ⊃
          pm`states(state[State, Unit](memory_write(pm`mem)(addr, byte)(s)))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_states_memory_write, matching in *,

This completes the proof of `allocator_stays_unchanged_memory_write_private_mem_TCC2`.
Q.E.D.

C.10.38 Allocator.allocator_stays_unchanged_memory_write_private_mem

Terse proof for `allocator_stays_unchanged_memory_write_private_mem`.
`allocator_stays_unchanged_memory_write_private_mem`:

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, byte: Byte, s: State):
pm‘states(s) ∧ pm‘rw_addr(addr) ∧ ¬ ac‘private_mem(pm)(addr) ⊃
stays_unchanged(pm)
(s, state(memory_write(pm‘mem)(addr, byte)(s)), ac‘private_mem(pm))

Installing automatic rewrites from: plain_memory_states_memory_write
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_stays_unchanged_memory_write,
Simplifying, rewriting, and recording with decision procedures,
Using lemma stays_unchanged_mono,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -1, 3,
Using lemma allocator_subset_private_memory_plain_memory,
Keeping (-1 1 2) and hiding *,
Rewriting relative to the theory: sets,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of `allocator_stays_unchanged_memory_write_private_mem`.
Q.E.D.

C.10.39 Allocator.allocator_allocated_memory_write

Terse proof for `allocator_allocated_memory_write`.
`allocator_allocated_memory_write`:

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, byte: Byte, s: State):
pm‘states(s) ∧ pm‘rw_addr(addr) ∧ ¬ ac‘private_mem(pm)(addr) ⊃
ac‘allocated(pm)(state(memory_write(pm‘mem)(addr, byte)(s))) =
ac‘allocated(pm)(s)

Installing automatic rewrites from: plain_memory_states_memory_write
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_const_private_memory_allocated, matching in *,
Rewriting using stays_unchanged_symmetric, matching in *,
Rewriting using allocator_stays_unchanged_memory_write_private_mem, matching in *,
This completes the proof of `allocator_allocated_memory_write`.
Q.E.D.

C.10.40 Allocator.allocator_memory_pool_memory_write

Terse proof for `allocator_memory_pool_memory_write`.

`allocator_memory_pool_memory_write:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), addr: Address, byte: Byte, s: State):
pm‘states(s) ∧ pm‘rw_addr(addr) ∧ ¬ ac‘private_mem(pm)(addr) ⊃
ac‘memory_pool(pm)(state(memory_write(pm‘mem)(addr, byte)(s))) =
ac‘memory_pool(pm)(s)

Installing automatic rewrites from: plain_memory_states_memory_write
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_const_private_memory_memory_pool, matching in *,
Rewriting using stays_unchanged_symmetric, matching in *,
Rewriting using allocator_stays_unchanged_memory_write_private_mem, matching in *,
This completes the proof of `allocator_memory_pool_memory_write`.
Q.E.D.

C.10.41 Allocator.allocator_subset_address_block_alloc_memory_pool

Terse proof for `allocator_subset_address_block_alloc_memory_pool`.

`allocator_subset_address_block_alloc_memory_pool:`

{1} ∀ (pm: (plain_memory?), ac: (allocator?(pm)), size: posnat, addr: Address, s: State):
pm‘states(s) ∧
OK?(ac‘alloc(pm)(size)(s)) ∧ ¬ data(ac‘alloc(pm)(size)(s)) = null_address
⊃
(address_block(data(ac‘alloc(pm)(size)(s)), size) ⊆ ac‘memory_pool(pm)(s))

Repeatedly Skolemizing and flattening,
Using lemma allocator_subset_address_block_alloc_allocated,
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_next_state_alloc,
Installing automatic rewrites from: has_next_state
Simplifying, rewriting, and recording with decision procedures,
Using lemma allocator_subset_allocated_memory_pool,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using allocator_memory_pool_alloc, matching in *,
Forward chaining on subset_transitive,
This completes the proof of `allocator_subset_address_block_alloc_memory_pool`.
Q.E.D.

C.11 Proofs for ArithmeticExpressions (expressions.pvs)

C.11.1 ArithmeticExpressions.mod_impl_TCC1

Terse proof for `mod_impl_TCC1`.

mod_impl_TCC1:

{1} ∃ (x: ⌊d: ⌊int, {k: int | k ≠ 0}⌋ → {r: int | abs(r) < abs(d‘2)}⌋): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (a: int, b: {k: int | k ≠ 0}): 0),
we get 2 subgoals:
mod_impl_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of mod_impl_TCC1.1.
mod_impl_TCC1.2:

{1} ∀ (b: {k: int | k ≠ 0}): abs(0) < abs(b)

Repeatedly Skolemizing and flattening,
Expanding the definition of abs,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of mod_impl_TCC1.2.
Q.E.D.

C.11.2 ArithmeticExpressions.div_impl_pos_range_TCC1

Terse proof for div_impl_pos_range_TCC1.
div_impl_pos_range_TCC1:

{1} ∀ (j: nat): j ≠ 0 ⊃ j > 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of div_impl_pos_range_TCC1.
Q.E.D.

C.11.3 ArithmeticExpressions.plus_ptr_TCC1

Terse proof for plus_ptr_TCC1.
plus_ptr_TCC1:

{1} ∀ (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
ptr_val: (range_pointer(cv_base(typ))), n: (range(i_typ))):
not_null?(typ)(ptr_val) ⊃
array?(cv_base(typ)) ∨
pointer?(cv_base(typ)) ∨
reference?(cv_base(typ)) ∨
bitfield?(cv_base(typ)) ∨
enum?(cv_base(typ)) ∨
pointer_to_member?(cv_base(typ)) ∨
const?(cv_base(typ)) ∨ volatile?(cv_base(typ))

Repeatedly Skolemizing and flattening,
Keeping (-2 2) and hiding *,

Rewriting using cv_base_result, matching in *,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plus_ptr_TCC1`.

Q.E.D.

C.11.4 ArithmeticExpressions.plus_ptr_TCC2

Terse proof for `plus_ptr_TCC2`.

`plus_ptr_TCC2:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
         ptr_val: (range_pointer(cv_base(typ))), n: (range(i_typ))):
       not_null?(typ)(ptr_val) ⊃ Cpp_Type?(typ(cv_base(typ)))
```

Repeatedly Skolemizing and flattening,

Keeping (-1 -2 1) and hiding *,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

plus_ptr_TCC2.1:

{-1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

{-2} $\text{cv(pointer?)}(\text{typ}')$

{-3} $\text{subterm}(t', \text{typ}(\text{cv\_base}(\text{typ}')))$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

which is trivially true.
This completes the proof of `plus_ptr_TCC2.1`.
`plus_ptr_TCC2.2:`

{-1} cv(pointer?)(typ′)
{-2} subterm(t′, typ(cv_base(typ′)))

{1} subterm(t′, typ′)
{2} no_pointers_to_bitfield?(t′) ∧
no_pointers_to_references?(t′) ∧
no_cv_references?(t′) ∧
no_reference_to_reference?(t′) ∧
no_reference_to_bitfields?(t′) ∧
no_pointer_to_member_to_reference?(t′) ∧
no_pointer_to_member_to_cv_void?(t′) ∧
no_cv_void_parameter?(t′) ∧
no_array_of_references?(t′) ∧
no_array_of_cv_void?(t′) ∧
no_array_of_function?(t′) ∧
no_array_of_abstract_class?(t′) ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t′) ∧
no_array_return_type?(t′) ∧
no_function_return_type?(t′) ∧
bitfield_underlying_integral_or_enum_type?(t′) ∧
cv_array?(t′) ∧
enum_underlying_integral?(t′) ∧
enum_constants?(t′) ∧
const_volatile?(t′) ∧
const_stutter?(t′) ∧
volatile_stutter?(t′) ∧
no_cv_class?(t′) ∧
no_cv_union?(t′) ∧
no_cv_function?(t′) ∧
no_reference_to_void?(t′) ∧
no_cv_void?(t′) ∧ no_array_of_bitfields?(t′)

Hiding formulas: 2,
Using lemma subterm_cv_base,
Using lemma cv_base_result,
we get 2 subgoals:
`plus_ptr_TCC2.2.1:`

{-1} pointer?(cv_base(typ′))
{-2} subterm(cv_base(typ′), typ′)
{-3} cv(pointer?)(typ′)
{-4} subterm(t′, typ(cv_base(typ′)))

{1} subterm(t′, typ′)

Using lemma subterm_transitive,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -4 1) and hiding *,
Expanding the definition of subterm,
which is trivially true.
This completes the proof of `plus_ptr_TCC2.2.1`.

`plus_ptr_TCC2.2.2:`

```
{-1}  subterm(cv_base(typ′), typ′)
{-2}  cv(pointer?)(typ′)
{-3}  subterm(t′, typ(cv_base(typ′)))
-------------------------------------
{1}   (pointer? ⊆ interpreted?)
{2}   subterm(t′, typ′)
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plus_ptr_TCC2.2.2`.
Q.E.D.

C.11.5 ArithmeticExpressions.plus_ptr_TCC3

Terse proof for `plus_ptr_TCC3`.

`plus_ptr_TCC3:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
         ptr_val: (range_pointer(cv_base(typ))), n: (range(i_typ))):
       not_null?(typ)(ptr_val) ⊃ pointer?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in * where P gets pointer?,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plus_ptr_TCC3`.
Q.E.D.

C.11.6 ArithmeticExpressions.plus_ptr_TCC4

Terse proof for `plus_ptr_TCC4`.

`plus_ptr_TCC4:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
         ptr_val: (range_pointer(cv_base(typ))), n: (range(i_typ))):
       ¬ (not_null?(typ)(ptr_val) ∧
            range_pointer(cv_base(typ))
                          (add(typ)(ptr_val, n × size_of(typ(cv_base(typ)))))
            ∧
            check_bounds(typ)
                          (add(typ)(ptr_val, n × size_of(typ(cv_base(typ))))))
        ⊃ pointer?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plus_ptr_TCC4`.
Q.E.D.

C.11.7 ArithmeticExpressions.plus_ptr_TCC5

Terse proof for `plus_ptr_TCC5`.
`plus_ptr_TCC5:`

```
{1}   ∀ (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
            ptr_val: (range_pointer(cv_base(typ)))):
        pointer?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plus_ptr_TCC5`.
Q.E.D.

C.11.8 ArithmeticExpressions.plus_ptr_TCC6

Terse proof for `plus_ptr_TCC6`.
`plus_ptr_TCC6:`

```
{1}   ∀ (typ: Cpp_Subtype(cv(pointer?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
        pointer?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plus_ptr_TCC6`.
Q.E.D.

C.11.9 ArithmeticExpressions.postinc_ptr_TCC1

Terse proof for `postinc_ptr_TCC1`.
`postinc_ptr_TCC1:`

```
{1}   ∀ ((typ: Cpp_Subtype(v(pointer?)))): cv(pointer?)(typ)
```

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `postinc_ptr_TCC1`.
Q.E.D.

C.11.10 ArithmeticExpressions.postinc_ptr_TCC2

Terse proof for `postinc_ptr_TCC2`.
`postinc_ptr_TCC2:`

```
{1}   ∀ ((typ: Cpp_Subtype(v(pointer?)))): pointer?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Using lemma postinc_ptr_TCC1,

Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `postinc_ptr_TCC2`.
Q.E.D.

C.11.11 ArithmeticExpressions.postinc_ptr_TCC3

Terse proof for `postinc_ptr_TCC3`.
`postinc_ptr_TCC3:`

{1} ∀ ((typ: Cpp_Subtype(*v*(pointer?)))): range(uchar)(1)

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Expanding the definition of range,
Expanding the definition of range_integral,
Expanding the definition of extend,
Using lemma binary_range_uchar,
Using lemma max_uchar,
Using lemma pos_expt_gt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `postinc_ptr_TCC3`.
Q.E.D.

C.11.12 ArithmeticExpressions.postinc_ptr_TCC4

Terse proof for `postinc_ptr_TCC4`.
`postinc_ptr_TCC4:`

{1} ∀ ((typ: Cpp_Subtype(*v*(pointer?)))):
Cpp_Type?(uchar) ∧ cv(non_bool_integral_enum?)(uchar)

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`postinc_ptr_TCC4.1:`

{1} Cpp_Type?(uchar)

Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `postinc_ptr_TCC4.1`.
`postinc_ptr_TCC4.2`:

```
{1}  cv(non_bool_integral_enum?)(uchar)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `postinc_ptr_TCC4.2`.
Q.E.D.

C.11.13 ArithmeticExpressions.subscript_TCC1

Terse proof for `subscript_TCC1`.
`subscript_TCC1`:

```
{1}  ∀ ((typ: Cpp_Subtype(array?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
       array?(typ) ∨
         pointer?(typ) ∨
           reference?(typ) ∨
             bitfield?(typ) ∨
               enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)
```

Repeatedly Skolemizing and flattening,
This completes the proof of `subscript_TCC1`.
Q.E.D.

C.11.14 ArithmeticExpressions.subscript_TCC2

Terse proof for `subscript_TCC2`.
`subscript_TCC2`:

```
{1}  ∀ ((typ: Cpp_Subtype(array?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
       Cpp_Type?(pointer(typ(typ))) ∧ cv(pointer?)(pointer(typ(typ)))
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
`subscript_TCC2.1`:

```
{-1}  Cpp_Type?(typ′)
{-2}  array?(typ′)
{-3}  Cpp_Type?(i_typ′)
{-4}  cv(non_bool_integral_enum?)(i_typ′)
{1}   Cpp_Type?(pointer(typ(typ′)))
```

Keeping (-1 -2 1) and hiding *,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Splitting conjunctions,
we get 2 subgoals:

`subscript_TCC2.1.1:`

{-1} $t' = \text{pointer}(\text{typ}(\text{typ}'))$
{-2} $\forall$ (t: Cpp_Type_):
$\text{subterm}(t, \text{typ}') \supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)
{-3} array?(typ')

{1} no_pointers_to_bitfield?(t') $\wedge$
no_pointers_to_references?(t') $\wedge$
no_cv_references?(t') $\wedge$
no_reference_to_reference?(t') $\wedge$
no_reference_to_bitfields?(t') $\wedge$
no_pointer_to_member_to_reference?(t') $\wedge$
no_pointer_to_member_to_cv_void?(t') $\wedge$
no_cv_void_parameter?(t') $\wedge$
no_array_of_references?(t') $\wedge$
no_array_of_cv_void?(t') $\wedge$
no_array_of_function?(t') $\wedge$
no_array_of_abstract_class?(t') $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t') $\wedge$
no_array_return_type?(t') $\wedge$
no_function_return_type?(t') $\wedge$
bitfield_underlying_integral_or_enum_type?(t') $\wedge$
cv_array?(t') $\wedge$
enum_underlying_integral?(t') $\wedge$
enum_constants?(t') $\wedge$
const_volatile?(t') $\wedge$
const_stutter?(t') $\wedge$
volatile_stutter?(t') $\wedge$
no_cv_class?(t') $\wedge$
no_cv_union?(t') $\wedge$
no_cv_function?(t') $\wedge$
no_reference_to_void?(t') $\wedge$
no_cv_void?(t') $\wedge$ no_array_of_bitfields?(t')

Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_array_of_bitfields? no_pointer_or_ref_to_incomplete_array_pa no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Instantiating the top quantifier in -2 with the terms: (typ!1),

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `subscript_TCC2.1.1`.

subscript_TCC2.1.2:

{-1} subterm(t', typ(typ$'$))
{-2} $\forall$ (t: Cpp_Type_):
subterm(t, typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)
{-3} array?(typ$'$)

{1} no_pointers_to_bitfield?(t') $\wedge$
no_pointers_to_references?(t') $\wedge$
no_cv_references?(t') $\wedge$
no_reference_to_reference?(t') $\wedge$
no_reference_to_bitfields?(t') $\wedge$
no_pointer_to_member_to_reference?(t') $\wedge$
no_pointer_to_member_to_cv_void?(t') $\wedge$
no_cv_void_parameter?(t') $\wedge$
no_array_of_references?(t') $\wedge$
no_array_of_cv_void?(t') $\wedge$
no_array_of_function?(t') $\wedge$
no_array_of_abstract_class?(t') $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t') $\wedge$
no_array_return_type?(t') $\wedge$
no_function_return_type?(t') $\wedge$
bitfield_underlying_integral_or_enum_type?(t') $\wedge$
cv_array?(t') $\wedge$
enum_underlying_integral?(t') $\wedge$
enum_constants?(t') $\wedge$
const_volatile?(t') $\wedge$
const_stutter?(t') $\wedge$
volatile_stutter?(t') $\wedge$
no_cv_class?(t') $\wedge$
no_cv_union?(t') $\wedge$
no_cv_function?(t') $\wedge$
no_reference_to_void?(t') $\wedge$
no_cv_void?(t') $\wedge$ no_array_of_bitfields?(t')

Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subscript_TCC2.1.2`.
`subscript_TCC2.2`:

{-1}	Cpp_Type?(typ′)
{-2}	array?(typ′)
{-3}	Cpp_Type?(i_typ′)
{-4}	cv(non_bool_integral_enum?)(i_typ′)
{1}	cv(pointer?)(pointer(typ(typ′)))

Expanding the definition of cv,
Expanding the definition of c,
which is trivially true.
This completes the proof of `subscript_TCC2.2`.
Q.E.D.

C.11.15 ArithmeticExpressions.subscript_TCC3

Terse proof for `subscript_TCC3`.
`subscript_TCC3`:

{1}	∀ ((typ: Cpp_Subtype(array?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))): Cpp_Type?(pointer(typ(typ)))

Repeatedly Skolemizing and flattening,
Using lemma subscript_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subscript_TCC3`.
Q.E.D.

C.11.16 ArithmeticExpressions.subscript_TCC4

Terse proof for `subscript_TCC4`.
`subscript_TCC4`:

{1}	∀ ((typ: Cpp_Subtype(array?)), i_typ: Cpp_Subtype(cv(non_bool_integral_enum?))): pointer?(cv_base(pointer(typ(typ))))

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result2, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subscript_TCC4`.
Q.E.D.

C.11.17 ArithmeticExpressions.ptr_to_nonzero_object_TCC1

Terse proof for `ptr_to_nonzero_object_TCC1`.

`ptr_to_nonzero_object_TCC1:`

```
{1}  ∀ (typ:
           Cpp_Subtype(extend[Cpp_Type_, (pointer?), bool, FALSE]
                             (λ (p: (pointer?)):
                                 ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))):
       array?(cv_base(typ)) ∨
        pointer?(cv_base(typ)) ∨
         reference?(cv_base(typ)) ∨
          bitfield?(cv_base(typ)) ∨
           enum?(cv_base(typ)) ∨
            pointer_to_member?(cv_base(typ)) ∨
             const?(cv_base(typ)) ∨ volatile?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 4) and hiding *,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:
`ptr_to_nonzero_object_TCC1.1:`

```
{-1}  pointer?(typ′)
{1}   cv(pointer?)(typ′)
{2}   pointer?(cv_base(typ′))
```

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ptr_to_nonzero_object_TCC1.1`.
`ptr_to_nonzero_object_TCC1.2:`

```
{-1}  pointer?(typ′)
{1}   (pointer? ⊆ interpreted?)
{2}   pointer?(cv_base(typ′))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ptr_to_nonzero_object_TCC1.2`.
Q.E.D.

C.11.18 ArithmeticExpressions.ptr_to_nonzero_object_TCC2

Terse proof for `ptr_to_nonzero_object_TCC2`.
`ptr_to_nonzero_object_TCC2:`

```
{1}  ∀ (typ:
           Cpp_Subtype(extend[Cpp_Type_, (pointer?), bool, FALSE]
                             (λ (p: (pointer?)):
                                 ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))):
       array?(typ(cv_base(typ))) ⊃ Cpp_Type?(typ(cv_base(typ)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,
Keeping (-1 -2 3) and hiding *,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ptr_to_nonzero_object_TCC2`.
Q.E.D.

C.11.19 ArithmeticExpressions.ptr_to_nonzero_object_TCC3

Terse proof for `ptr_to_nonzero_object_TCC3`.
`ptr_to_nonzero_object_TCC3:`

```
{1}   ∀ (typ:
              Cpp_Subtype(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))):
        (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ⊃
         Cpp_Type?(typ(cv_base(typ)))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -3,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
Keeping (-1 -2 3) and hiding *,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ptr_to_nonzero_object_TCC3`.
Q.E.D.

C.11.20 ArithmeticExpressions.ptr_to_nonzero_object

Terse proof for `ptr_to_nonzero_object`.
`ptr_to_nonzero_object:`

```
{1}   ∀ (typ:
              Cpp_Subtype(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))):
        (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ⊃
         size_of(typ(cv_base(typ))) > 0
```

Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,

Expanding the definition of size_of,
Expanding the definition of uidt,
Expanding the definition of Cpp_Type?,
Instantiating the top quantifier in -1 with the terms: (typ!1),
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Hiding formulas: (-4 -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16),
Hiding formulas: (-13 -15),
Expanding the definition of no_pointers_to_bitfield?,
Expanding the definition of no_pointers_to_references?,
Installing automatic rewrites from: pod_size
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 13 subgoals:
`ptr_to_nonzero_object.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ′)))) > 0

Expanding the definition of dt_cv_pointer,
Adding type constraints for dt_pointer(typ(typ!1)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ptr_to_nonzero_object.1`.
`ptr_to_nonzero_object.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer_to_member?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_ptm(typ(typ′)))) > 0

Expanding the definition of dt_cv_ptm,

Adding type constraints for dt_ptm(typ(typ!1)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.2`.

`ptr_to_nonzero_object.3:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	bitfield?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_bitfield(typ(typ′)))) > 0

Expanding the definition of cv,

Expanding the definition of c,

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.3`.

`ptr_to_nonzero_object.4:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(typ(typ(typ′)))) > 0

Expanding the definition of uidt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

`ptr_to_nonzero_object.4.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer?(typ(typ(typ(typ′))))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_pointer,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.4.1.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer?(typ(typ(typ(typ′))))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ(typ′)))))) > 0

Expanding the definition of dt_cv_pointer,

Adding type constraints for dt_pointer(typ(typ(typ(typ!1)))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.1.1`.

`ptr_to_nonzero_object.4.1.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer?(typ(typ(typ(typ′))))
{1}	∃ (x_1: (interpreted_data_type? [((range_pointer(cv_base(typ(typ(typ(typ′)))))))])): TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ(typ′))))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(typ(typ(typ(typ!1))))))),

we get 2 subgoals:

`ptr_to_nonzero_object.4.1.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer?(typ(typ(typ(typ′))))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ(typ′))))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.4.1.2.1`.

`ptr_to_nonzero_object.4.1.2.2:`

```
{-1}  no_cv_references?(typ')
{-2}  cv_array?(typ')
{-3}  enum_underlying_integral?(typ')
{-4}  enum_constants?(typ')
{-5}  const_volatile?(typ')
{-6}  const_stutter?(typ')
{-7}  volatile_stutter?(typ')
{-8}  no_cv_class?(typ')
{-9}  no_cv_union?(typ')
{-10} no_cv_function?(typ')
{-11} no_cv_void?(typ')
{-12} pointer?(typ')
{-13} const?(typ(typ'))
{-14} volatile?(typ(typ(typ')))
{-15} pointer?(typ(typ(typ(typ'))))
------------------------------------------------------------------------
 {1}  pointer?(cv_base(typ(typ(typ(typ')))))
 {2}  cv(bitfield?)(typ(typ'))
 {3}  size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ(typ')))))))  >  0
```

Keeping (-15 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.1.2.2`.

`ptr_to_nonzero_object.4.2:`

```
{-1}  no_cv_references?(typ')
{-2}  cv_array?(typ')
{-3}  enum_underlying_integral?(typ')
{-4}  enum_constants?(typ')
{-5}  const_volatile?(typ')
{-6}  const_stutter?(typ')
{-7}  volatile_stutter?(typ')
{-8}  no_cv_class?(typ')
{-9}  no_cv_union?(typ')
{-10} no_cv_function?(typ')
{-11} no_cv_void?(typ')
{-12} pointer?(typ')
{-13} const?(typ(typ'))
{-14} volatile?(typ(typ(typ')))
{-15} pointer_to_member?(typ(typ(typ(typ'))))
------------------------------------------------------------------------
 {1}  cv(bitfield?)(typ(typ'))
 {2}  size(uidt(dt_cv_ptm(typ(typ(typ')))))  >  0
```

Expanding the definition of dt_cv_ptm,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.4.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer_to_member?(typ(typ(typ(typ′))))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_ptm(typ(typ(typ(typ′)))))) > 0

Expanding the definition of dt_cv_ptm,

Adding type constraints for dt_ptm(typ(typ(typ(typ!1)))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.2.1`.

`ptr_to_nonzero_object.4.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer_to_member?(typ(typ(typ(typ′))))
{1}	∃ (x_1 : (interpreted_data_type?[((range_ptm(cv_base(typ(typ(typ(typ′)))))))])) : TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ(typ′)))))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_ptm(cv_base(typ(typ(typ(typ!1)))))),

we get 2 subgoals:

`ptr_to_nonzero_object.4.2.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer_to_member?(typ(typ(typ(typ′))))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ(typ′))))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.4.2.2.1`.

`ptr_to_nonzero_object.4.2.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	pointer_to_member?(typ(typ(typ(typ′))))
{1}	pointer_to_member?(cv_base(typ(typ(typ(typ′)))))
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ(typ′))))))) > 0

Keeping (-15 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.2.2.2`.

`ptr_to_nonzero_object.4.3:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	bitfield?(typ(typ(typ(typ′))))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_bitfield(typ(typ(typ′))))) > 0

Expanding the definition of cv,

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.4.3`.

`ptr_to_nonzero_object.4.4:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	floating_point?(typ(typ(typ(typ′))))
{1}	cv(bitfield?)(typ(typ′))
{2}	bool?(typ(typ(typ(typ′))))
{3}	pointer?(typ(typ(typ(typ′))))
{4}	pointer_to_member?(typ(typ(typ(typ′))))
{5}	bitfield?(typ(typ(typ(typ′))))
{6}	size(uidt(dt_cv_float(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_float,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.4.4.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	floating_point?(typ(typ(typ(typ′))))
{1}	cv(bitfield?)(typ(typ′))
{2}	bool?(typ(typ(typ(typ′))))
{3}	pointer?(typ(typ(typ(typ′))))
{4}	pointer_to_member?(typ(typ(typ(typ′))))
{5}	bitfield?(typ(typ(typ(typ′))))
{6}	size(uidt(dt_cv_float(typ(typ(typ(typ′)))))) > 0

Expanding the definition of dt_cv_float,

Adding type constraints for dt_floating_point(typ(typ(typ(typ!1)))),

Expanding the definition of floating_point?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.4.1`.

`ptr_to_nonzero_object.4.4.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	floating_point?(typ(typ(typ(typ′))))
{1}	∃ (x_1 : (interpreted_data_type? [((range_floating_point(cv_base(typ(typ(typ(typ′)))))))])) : TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ(typ′))))
{4}	pointer?(typ(typ(typ(typ′))))
{5}	pointer_to_member?(typ(typ(typ(typ′))))
{6}	bitfield?(typ(typ(typ(typ′))))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ(typ′))))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(typ(typ(typ(typ!1)))))), we get 2 subgoals:

`ptr_to_nonzero_object.4.4.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	floating_point?(typ(typ(typ(typ′))))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ(typ′))))
{4}	pointer?(typ(typ(typ(typ′))))
{5}	pointer_to_member?(typ(typ(typ(typ′))))
{6}	bitfield?(typ(typ(typ(typ′))))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ(typ′))))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.4.4.2.1`.

`ptr_to_nonzero_object.4.4.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	volatile?(typ(typ(typ′)))
{-15}	floating_point?(typ(typ(typ(typ′))))
{1}	floating_point?(cv_base(typ(typ(typ(typ′)))))
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ(typ′))))
{4}	pointer?(typ(typ(typ(typ′))))
{5}	pointer_to_member?(typ(typ(typ(typ′))))
{6}	bitfield?(typ(typ(typ(typ′))))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ(typ′))))))) > 0

Keeping (-15 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.4.4.2.2`.

`ptr_to_nonzero_object.5:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	floating_point?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	bool?(typ(typ(typ′)))
{3}	pointer?(typ(typ(typ′)))
{4}	pointer_to_member?(typ(typ(typ′)))
{5}	bitfield?(typ(typ(typ′)))
{6}	volatile?(typ(typ(typ′)))
{7}	size(uidt(dt_cv_float(typ(typ′)))) > 0

Expanding the definition of dt_cv_float,

Rewriting using dt_const_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.5.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	floating_point?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	bool?(typ(typ(typ′)))
{3}	pointer?(typ(typ(typ′)))
{4}	pointer_to_member?(typ(typ(typ′)))
{5}	bitfield?(typ(typ(typ′)))
{6}	volatile?(typ(typ(typ′)))
{7}	size(uidt(dt_cv_float(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_float,

Adding type constraints for dt_floating_point(typ(typ(typ!1))),

Expanding the definition of floating_point?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.5.1`.

`ptr_to_nonzero_object.5.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	floating_point?(typ(typ(typ′)))
{1}	∃ (x_1 : (interpreted_data_type? [((range_floating_point(cv_base(typ(typ(typ′))))))])): TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ′)))
{4}	pointer?(typ(typ(typ′)))
{5}	pointer_to_member?(typ(typ(typ′)))
{6}	bitfield?(typ(typ(typ′)))
{7}	volatile?(typ(typ(typ′)))
{8}	size(uidt(dt_const(dt_cv_float(typ(typ(typ′)))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(typ(typ(typ!1))))), we get 2 subgoals:

`ptr_to_nonzero_object.5.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	floating_point?(typ(typ(typ′)))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ′)))
{4}	pointer?(typ(typ(typ′)))
{5}	pointer_to_member?(typ(typ(typ′)))
{6}	bitfield?(typ(typ(typ′)))
{7}	volatile?(typ(typ(typ′)))
{8}	size(uidt(dt_const(dt_cv_float(typ(typ(typ′)))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.5.2.1`.

`ptr_to_nonzero_object.5.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	floating_point?(typ(typ(typ′)))
{1}	floating_point?(cv_base(typ(typ(typ′))))
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ′)))
{4}	pointer?(typ(typ(typ′)))
{5}	pointer_to_member?(typ(typ(typ′)))
{6}	bitfield?(typ(typ(typ′)))
{7}	volatile?(typ(typ(typ′)))
{8}	size(uidt(dt_const(dt_cv_float(typ(typ(typ′)))))) > 0

Keeping (-14 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.5.2.2`.

`ptr_to_nonzero_object.6:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	floating_point?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	reference?(typ(typ′))
{3}	void?(typ(typ′))
{4}	function?(typ(typ′))
{5}	array?(typ(typ′))
{6}	class?(typ(typ′))
{7}	union?(typ(typ′))
{8}	pointer?(typ(typ′))
{9}	pointer_to_member?(typ(typ′))
{10}	bitfield?(typ(typ′))
{11}	bool?(typ(typ′))
{12}	const?(typ(typ′))
{13}	size(uidt(dt_cv_float(typ(typ′)))) > 0
{14}	volatile?(typ(typ′))

Expanding the definition of dt_cv_float,

Adding type constraints for dt_floating_point(typ(typ!1)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.6`.

`ptr_to_nonzero_object.7:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	floating_point?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	bool?(typ(typ(typ′)))
{3}	pointer?(typ(typ(typ′)))
{4}	pointer_to_member?(typ(typ(typ′)))
{5}	bitfield?(typ(typ(typ′)))
{6}	size(uidt(dt_cv_float(typ(typ′)))) > 0

Expanding the definition of dt_cv_float,
Rewriting using dt_volatile_size, matching in *,
we get 2 subgoals:
`ptr_to_nonzero_object.7.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	floating_point?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	bool?(typ(typ(typ′)))
{3}	pointer?(typ(typ(typ′)))
{4}	pointer_to_member?(typ(typ(typ′)))
{5}	bitfield?(typ(typ(typ′)))
{6}	size(uidt(dt_cv_float(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_float,
Adding type constraints for dt_floating_point(typ(typ(typ!1))),
Expanding the definition of floating_point?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ptr_to_nonzero_object.7.1`.

`ptr_to_nonzero_object.7.2:`

```
{-1}  no_cv_references?(typ′)
{-2}  cv_array?(typ′)
{-3}  enum_underlying_integral?(typ′)
{-4}  enum_constants?(typ′)
{-5}  const_volatile?(typ′)
{-6}  const_stutter?(typ′)
{-7}  volatile_stutter?(typ′)
{-8}  no_cv_class?(typ′)
{-9}  no_cv_union?(typ′)
{-10} no_cv_function?(typ′)
{-11} no_cv_void?(typ′)
{-12} pointer?(typ′)
{-13} floating_point?(typ(typ(typ′)))
{-14} volatile?(typ(typ′))
-------------------------------------------------------------------
 {1}  ∃ (x1 :
           (interpreted_data_type?
                [((range_floating_point(cv_base(typ(typ(typ′))))))])):
        TRUE
 {2}  cv(bitfield?)(typ(typ′))
 {3}  bool?(typ(typ(typ′)))
 {4}  pointer?(typ(typ(typ′)))
 {5}  pointer_to_member?(typ(typ(typ′)))
 {6}  bitfield?(typ(typ(typ′)))
 {7}  size(uidt(dt_volatile(dt_cv_float(typ(typ(typ′)))))) > 0
```

Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(typ(typ(typ!1))))), we get 2 subgoals:

`ptr_to_nonzero_object.7.2.1:`

```
{-1}  no_cv_references?(typ′)
{-2}  cv_array?(typ′)
{-3}  enum_underlying_integral?(typ′)
{-4}  enum_constants?(typ′)
{-5}  const_volatile?(typ′)
{-6}  const_stutter?(typ′)
{-7}  volatile_stutter?(typ′)
{-8}  no_cv_class?(typ′)
{-9}  no_cv_union?(typ′)
{-10} no_cv_function?(typ′)
{-11} no_cv_void?(typ′)
{-12} pointer?(typ′)
{-13} floating_point?(typ(typ(typ′)))
{-14} volatile?(typ(typ′))
-------------------------------------------------------------------
 {1}  TRUE
 {2}  cv(bitfield?)(typ(typ′))
 {3}  bool?(typ(typ(typ′)))
 {4}  pointer?(typ(typ(typ′)))
 {5}  pointer_to_member?(typ(typ(typ′)))
 {6}  bitfield?(typ(typ(typ′)))
 {7}  size(uidt(dt_volatile(dt_cv_float(typ(typ(typ′)))))) > 0
```

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.7.2.1`.
`ptr_to_nonzero_object.7.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	floating_point?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	floating_point?(cv_base(typ(typ(typ′))))
{2}	cv(bitfield?)(typ(typ′))
{3}	bool?(typ(typ(typ′)))
{4}	pointer?(typ(typ(typ′)))
{5}	pointer_to_member?(typ(typ(typ′)))
{6}	bitfield?(typ(typ(typ′)))
{7}	size(uidt(dt_volatile(dt_cv_float(typ(typ(typ′)))))) > 0

Keeping (-13 1) and hiding *,
Expanding the definition of cv_base,
Expanding the definition of floating_point?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ptr_to_nonzero_object.7.2.2`.
`ptr_to_nonzero_object.8:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	bitfield?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_bitfield(typ(typ′)))) > 0

Expanding the definition of cv,
Expanding the definition of c,
which is trivially true.
This completes the proof of `ptr_to_nonzero_object.8`.

`ptr_to_nonzero_object.9:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	bitfield?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_bitfield(typ(typ′)))) > 0

Expanding the definition of cv,

Expanding the definition of c,

Expanding the definition of v,

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.9`.

`ptr_to_nonzero_object.10:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer_to_member?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_ptm(typ(typ′)))) > 0

Expanding the definition of dt_cv_ptm,

Rewriting using dt_const_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.10.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer_to_member?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_ptm(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_ptm,

Adding type constraints for dt_ptm(typ(typ(typ!1))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.10.1`.

`ptr_to_nonzero_object.10.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer_to_member?(typ(typ(typ′)))
{1}	∃ (x_1 : (interpreted_data_type?[((range_ptm(cv_base(typ(typ(typ′))))))])) : TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_const(dt_cv_ptm(typ(typ(typ′)))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_ptm(cv_base(typ(typ(typ!1))))),

we get 2 subgoals:

`ptr_to_nonzero_object.10.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer_to_member?(typ(typ(typ′)))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_const(dt_cv_ptm(typ(typ(typ′)))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.10.2.1`.

`ptr_to_nonzero_object.10.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer_to_member?(typ(typ(typ′)))
{1}	pointer_to_member?(cv_base(typ(typ(typ′))))
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_const(dt_cv_ptm(typ(typ(typ′)))))) > 0

Keeping (-14 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.10.2.2`.

`ptr_to_nonzero_object.11:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer_to_member?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_ptm(typ(typ′)))) > 0

Expanding the definition of dt_cv_ptm,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.11.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer_to_member?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_ptm(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_ptm,

Adding type constraints for dt_ptm(typ(typ(typ!1))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.11.1`.

`ptr_to_nonzero_object.11.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer_to_member?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	∃ (x_1 : (interpreted_data_type?[((range_ptm(cv_base(typ(typ(typ′)))))))])): TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ′)))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_ptm(cv_base(typ(typ(typ!1))))),

we get 2 subgoals:

`ptr_to_nonzero_object.11.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer_to_member?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ′)))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.11.2.1`.

`ptr_to_nonzero_object.11.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer_to_member?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	pointer_to_member?(cv_base(typ(typ(typ′))))
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_ptm(typ(typ(typ′)))))) > 0

Keeping (-13 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.11.2.2`.

`ptr_to_nonzero_object.12:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ′)))) > 0

Expanding the definition of dt_cv_pointer,

Rewriting using dt_const_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.12.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer?(typ(typ(typ′)))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_pointer,

Adding type constraints for dt_pointer(typ(typ(typ!1))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.12.1`.

`ptr_to_nonzero_object.12.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer?(typ(typ(typ′)))
{1}	$\exists$ (x_1 : (interpreted_data_type?$\lceil$((range_pointer(cv_base(typ(typ(typ′))))))$\rceil$)): TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_const(dt_cv_pointer(typ(typ(typ′)))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(typ(typ(typ!1))))),

we get 2 subgoals:

`ptr_to_nonzero_object.12.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer?(typ(typ(typ′)))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_const(dt_cv_pointer(typ(typ(typ′)))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.12.2.1`.

`ptr_to_nonzero_object.12.2.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	const?(typ(typ′))
{-14}	pointer?(typ(typ(typ′)))
{1}	pointer?(cv_base(typ(typ(typ′))))
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_const(dt_cv_pointer(typ(typ(typ′)))))) > 0

Keeping (-14 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.12.2.2`.

`ptr_to_nonzero_object.13:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ′)))) > 0

Expanding the definition of dt_cv_pointer,

Rewriting using dt_volatile_size, matching in *,

we get 2 subgoals:

`ptr_to_nonzero_object.13.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	cv(bitfield?)(typ(typ′))
{2}	size(uidt(dt_cv_pointer(typ(typ(typ′))))) > 0

Expanding the definition of dt_cv_pointer,

Adding type constraints for dt_pointer(typ(typ(typ!1))),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ptr_to_nonzero_object.13.1`.

`ptr_to_nonzero_object.13.2:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	∃ (x_1 : (interpreted_data_type?⌊((range_pointer(cv_base(typ(typ(typ′))))))⌋)): TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ′)))))) > 0

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(typ(typ(typ!1))))),

we get 2 subgoals:

`ptr_to_nonzero_object.13.2.1:`

{-1}	no_cv_references?(typ′)
{-2}	cv_array?(typ′)
{-3}	enum_underlying_integral?(typ′)
{-4}	enum_constants?(typ′)
{-5}	const_volatile?(typ′)
{-6}	const_stutter?(typ′)
{-7}	volatile_stutter?(typ′)
{-8}	no_cv_class?(typ′)
{-9}	no_cv_union?(typ′)
{-10}	no_cv_function?(typ′)
{-11}	no_cv_void?(typ′)
{-12}	pointer?(typ′)
{-13}	pointer?(typ(typ(typ′)))
{-14}	volatile?(typ(typ′))
{1}	TRUE
{2}	cv(bitfield?)(typ(typ′))
{3}	size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ′)))))) > 0

which is trivially true.

This completes the proof of `ptr_to_nonzero_object.13.2.1`.

`ptr_to_nonzero_object.13.2.2:`

```
{-1}   no_cv_references?(typ′)
{-2}   cv_array?(typ′)
{-3}   enum_underlying_integral?(typ′)
{-4}   enum_constants?(typ′)
{-5}   const_volatile?(typ′)
{-6}   const_stutter?(typ′)
{-7}   volatile_stutter?(typ′)
{-8}   no_cv_class?(typ′)
{-9}   no_cv_union?(typ′)
{-10}  no_cv_function?(typ′)
{-11}  no_cv_void?(typ′)
{-12}  pointer?(typ′)
{-13}  pointer?(typ(typ(typ′)))
{-14}  volatile?(typ(typ′))
-------------------------------------------------------------------------
 {1}   pointer?(cv_base(typ(typ(typ′))))
 {2}   cv(bitfield?)(typ(typ′))
 {3}   size(uidt(dt_volatile(dt_cv_pointer(typ(typ(typ′)))))) > 0
```

Keeping (-13 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ptr_to_nonzero_object.13.2.2`.
Q.E.D.

C.11.21 ArithmeticExpressions.minus_ptr_typ_is_Cpp_Type_TCC1

Terse proof for `minus_ptr_typ_is_Cpp_Type_TCC1`.

`minus_ptr_typ_is_Cpp_Type_TCC1:`

```
{1}  ∀ (typ:
          Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))):
       array?(cv_base(typ)) ∨
        pointer?(cv_base(typ)) ∨
         reference?(cv_base(typ)) ∨
          bitfield?(cv_base(typ)) ∨
           enum?(cv_base(typ)) ∨
            pointer_to_member?(cv_base(typ)) ∨
             const?(cv_base(typ)) ∨ volatile?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_typ_is_Cpp_Type_TCC1`.
Q.E.D.

C.11.22 ArithmeticExpressions.minus_ptr_typ_is_Cpp_Type

Terse proof for `minus_ptr_typ_is_Cpp_Type`.

`minus_ptr_typ_is_Cpp_Type:`

{1} ∀ (typ:
Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)):
¬ void?(typ(p)) ∧ ¬ function?(typ(p)))))):
Cpp_Type?(typ(cv_base(typ)))

Repeatedly Skolemizing and flattening,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -1 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

`minus_ptr_typ_is_Cpp_Type.1:`

{-1} $\mathrm{no\_pointers\_to\_bitfield?}(t') \wedge$
$\mathrm{no\_pointers\_to\_references?}(t') \wedge$
$\mathrm{no\_cv\_references?}(t') \wedge$
$\mathrm{no\_reference\_to\_reference?}(t') \wedge$
$\mathrm{no\_reference\_to\_bitfields?}(t') \wedge$
$\mathrm{no\_pointer\_to\_member\_to\_reference?}(t') \wedge$
$\mathrm{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge$
$\mathrm{no\_cv\_void\_parameter?}(t') \wedge$
$\mathrm{no\_array\_of\_references?}(t') \wedge$
$\mathrm{no\_array\_of\_cv\_void?}(t') \wedge$
$\mathrm{no\_array\_of\_function?}(t') \wedge$
$\mathrm{no\_array\_of\_abstract\_class?}(t') \wedge$
$\mathrm{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge$
$\mathrm{no\_array\_return\_type?}(t') \wedge$
$\mathrm{no\_function\_return\_type?}(t') \wedge$
$\mathrm{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge$
$\mathrm{cv\_array?}(t') \wedge$
$\mathrm{enum\_underlying\_integral?}(t') \wedge$
$\mathrm{enum\_constants?}(t') \wedge$
$\mathrm{const\_volatile?}(t') \wedge$
$\mathrm{const\_stutter?}(t') \wedge$
$\mathrm{volatile\_stutter?}(t') \wedge$
$\mathrm{no\_cv\_class?}(t') \wedge$
$\mathrm{no\_cv\_union?}(t') \wedge$
$\mathrm{no\_cv\_function?}(t') \wedge$
$\mathrm{no\_reference\_to\_void?}(t') \wedge$
$\mathrm{no\_cv\_void?}(t') \wedge \mathrm{no\_array\_of\_bitfields?}(t')$

{-2} $\mathrm{cv}(\mathrm{extend}[\mathrm{Cpp\_Type\_}, (\mathrm{pointer?}), \mathrm{bool}, \mathrm{FALSE}]$
$(\lambda\ (p\colon (\mathrm{pointer?}))\colon \neg\ \mathrm{void?}(\mathrm{typ}(p)) \wedge \neg\ \mathrm{function?}(\mathrm{typ}(p))))$
(typ')

{-3} $\mathrm{subterm}(t', \mathrm{typ}(\mathrm{cv\_base}(\mathrm{typ}')))$

{1} $\mathrm{no\_pointers\_to\_bitfield?}(t') \wedge$
$\mathrm{no\_pointers\_to\_references?}(t') \wedge$
$\mathrm{no\_cv\_references?}(t') \wedge$
$\mathrm{no\_reference\_to\_reference?}(t') \wedge$
$\mathrm{no\_reference\_to\_bitfields?}(t') \wedge$
$\mathrm{no\_pointer\_to\_member\_to\_reference?}(t') \wedge$
$\mathrm{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge$
$\mathrm{no\_cv\_void\_parameter?}(t') \wedge$
$\mathrm{no\_array\_of\_references?}(t') \wedge$
$\mathrm{no\_array\_of\_cv\_void?}(t') \wedge$
$\mathrm{no\_array\_of\_function?}(t') \wedge$
$\mathrm{no\_array\_of\_abstract\_class?}(t') \wedge$
$\mathrm{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge$
$\mathrm{no\_array\_return\_type?}(t') \wedge$
$\mathrm{no\_function\_return\_type?}(t') \wedge$
$\mathrm{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge$
$\mathrm{cv\_array?}(t') \wedge$
$\mathrm{enum\_underlying\_integral?}(t') \wedge$
$\mathrm{enum\_constants?}(t') \wedge$
$\mathrm{const\_volatile?}(t') \wedge$
$\mathrm{const\_stutter?}(t') \wedge$
$\mathrm{volatile\_stutter?}(t') \wedge$
$\mathrm{no\_cv\_class?}(t') \wedge$
$\mathrm{no\_cv\_union?}(t') \wedge$
$\mathrm{no\_cv\_function?}(t') \wedge$
$\mathrm{no\_reference\_to\_void?}(t') \wedge$
$\mathrm{no\_cv\_void?}(t') \wedge \mathrm{no\_array\_of\_bitfields?}(t')$

which is trivially true.

This completes the proof of `minus_ptr_typ_is_Cpp_Type.1`.

`minus_ptr_typ_is_Cpp_Type.2:`

{-1} cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
(typ′)
{-2} subterm(t′, typ(cv_base(typ′)))

{1} subterm(t′, typ′)
{2} no_pointers_to_bitfield?(t′) ∧
no_pointers_to_references?(t′) ∧
no_cv_references?(t′) ∧
no_reference_to_reference?(t′) ∧
no_reference_to_bitfields?(t′) ∧
no_pointer_to_member_to_reference?(t′) ∧
no_pointer_to_member_to_cv_void?(t′) ∧
no_cv_void_parameter?(t′) ∧
no_array_of_references?(t′) ∧
no_array_of_cv_void?(t′) ∧
no_array_of_function?(t′) ∧
no_array_of_abstract_class?(t′) ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t′) ∧
no_array_return_type?(t′) ∧
no_function_return_type?(t′) ∧
bitfield_underlying_integral_or_enum_type?(t′) ∧
cv_array?(t′) ∧
enum_underlying_integral?(t′) ∧
enum_constants?(t′) ∧
const_volatile?(t′) ∧
const_stutter?(t′) ∧
volatile_stutter?(t′) ∧
no_cv_class?(t′) ∧
no_cv_union?(t′) ∧
no_cv_function?(t′) ∧
no_reference_to_void?(t′) ∧
no_cv_void?(t′) ∧ no_array_of_bitfields?(t′)

Hiding formulas: 2,

Using lemma subterm_cv_base,

Using lemma subterm_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 1 2) and hiding *,

Using lemma cv_base_result,

we get 2 subgoals:

`minus_ptr_typ_is_Cpp_Type.2.1:`

```
{-1}  extend[Cpp_Type_, (pointer?), bool, FALSE]
        (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))
        (cv_base(typ′))
{-2}  cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
           (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
      (typ′)
{-3}  subterm(t′, typ(cv_base(typ′)))
-------------------------------------------------------------------------
{1}   subterm(t′, cv_base(typ′))
{2}   subterm(t′, typ′)
```

Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of subterm,
which is trivially true.
This completes the proof of `minus_ptr_typ_is_Cpp_Type.2.1`.
`minus_ptr_typ_is_Cpp_Type.2.2:`

```
{-1}  cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
           (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
      (typ′)
{-2}  subterm(t′, typ(cv_base(typ′)))
-------------------------------------------------------------------------
{1}   (extend [Cpp_Type_, (pointer?), bool, FALSE](λ (p : (pointer?)) : ¬ void?(typ(p)) ∧ ¬ function?(typ(p
{2}   subterm(t′, cv_base(typ′))
{3}   subterm(t′, typ′)
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_typ_is_Cpp_Type.2.2`.
Q.E.D.

C.11.23 ArithmeticExpressions.minus_ptr_ptr_TCC1

Terse proof for `minus_ptr_ptr_TCC1`.
`minus_ptr_ptr_TCC1:`

```
-------------------------------------------------------------------------
{1}   ∀ (typ:
          Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                           (λ (p: (pointer?)):
                              ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))),
            ptr_val1: (range_pointer(cv_base(typ))),
            ptr_val2: (range_pointer(cv_base(typ)))):
        array?(typ(cv_base(typ))) ⊃ Cpp_Type?(typ(cv_base(typ)))
```

Repeatedly Skolemizing and flattening,
Using lemma minus_ptr_typ_is_Cpp_Type,
This completes the proof of `minus_ptr_ptr_TCC1`.
Q.E.D.

C.11.24 ArithmeticExpressions.minus_ptr_ptr_TCC2

Terse proof for `minus_ptr_ptr_TCC2`.

minus_ptr_ptr_TCC2:

```
{1}  ∀ (typ:
           Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))),
             ptr_val1: (range_pointer(cv_base(typ))),
             ptr_val2: (range_pointer(cv_base(typ))):
         (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ⊃
          cv(pointer?)(typ)
```

Repeatedly Skolemizing and flattening,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of minus_ptr_ptr_TCC2.

Q.E.D.

C.11.25 ArithmeticExpressions.minus_ptr_ptr_TCC3

Terse proof for minus_ptr_ptr_TCC3.

minus_ptr_ptr_TCC3:

```
{1}  ∀ (typ:
           Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))),
             ptr_val1: (range_pointer(cv_base(typ))),
             ptr_val2: (range_pointer(cv_base(typ))):
         (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ∧
          not_null?(typ)(ptr_val1) ∧
           not_null?(typ)(ptr_val2) ∧
            same_array(typ)(ptr_val1, ptr_val2) ∧
             up?(address_of(typ)(ptr_val1)) ∧ up?(address_of(typ)(ptr_val2))
          ⊃ Cpp_Type?(typ(cv_base(typ)))
```

Repeatedly Skolemizing and flattening,

Using lemma minus_ptr_typ_is_Cpp_Type,

This completes the proof of minus_ptr_ptr_TCC3.

Q.E.D.

C.11.26 ArithmeticExpressions.minus_ptr_ptr_TCC4

Terse proof for minus_ptr_ptr_TCC4.

`minus_ptr_ptr_TCC4:`

{1} ∀ (typ:
Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)):
¬ void?(typ(p)) ∧ ¬ function?(typ(p)))),
ptr_val1: (range_pointer(cv_base(typ))),
ptr_val2: (range_pointer(cv_base(typ)))):
(array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ∧
not_null?(typ)(ptr_val1) ∧
not_null?(typ)(ptr_val2) ∧
same_array(typ)(ptr_val1, ptr_val2) ∧
up?(address_of(typ)(ptr_val1)) ∧ up?(address_of(typ)(ptr_val2))
⊃ size_of(typ(cv_base(typ))) > 0

Repeatedly Skolemizing and flattening,

Using lemma ptr_to_nonzero_object,

we get 2 subgoals:

`minus_ptr_ptr_TCC4.1:`

{-1} (array?(typ(cv_base(cv_base(typ′)))) ⊃
size_of(typ(cv_base(cv_base(typ′)))) > 0)
⊃ size_of(typ(cv_base(cv_base(typ′)))) > 0
{-2} Cpp_Type?(typ′)
{-3} cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
(typ′)
{-4} range_pointer(cv_base(typ′))(ptr_val1′)
{-5} range_pointer(cv_base(typ′))(ptr_val2′)
{-6} (array?(typ(cv_base(typ′))) ⊃ size_of(typ(cv_base(typ′))) > 0)
{-7} not_null?(typ′)(ptr_val1′)
{-8} not_null?(typ′)(ptr_val2′)
{-9} same_array(typ′)(ptr_val1′, ptr_val2′)
{-10} up?(address_of(typ′)(ptr_val1′))
{-11} up?(address_of(typ′)(ptr_val2′))

{1} size_of(typ(cv_base(typ′))) > 0

Expanding the definition of cv_base,

Using lemma cv_base_not_const,

Using lemma cv_base_not_volatile,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `minus_ptr_ptr_TCC4.1`.

`minus_ptr_ptr_TCC4.2`:

```
{-1}   Cpp_Type?(typ′)
{-2}   cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
            (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
         (typ′)
{-3}   range_pointer(cv_base(typ′))(ptr_val1′)
{-4}   range_pointer(cv_base(typ′))(ptr_val2′)
{-5}   (array?(typ(cv_base(typ′))) ⊃ size_of(typ(cv_base(typ′))) > 0)
{-6}   not_null?(typ′)(ptr_val1′)
{-7}   not_null?(typ′)(ptr_val2′)
{-8}   same_array(typ′)(ptr_val1′, ptr_val2′)
{-9}   up?(address_of(typ′)(ptr_val1′))
{-10}  up?(address_of(typ′)(ptr_val2′))
-------------------------------------------------------------------
 {1}   extend[Cpp_Type_, (pointer?), bool, FALSE]
         (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p)))
         (cv_base(typ′))
 {2}   size_of(typ(cv_base(typ′))) > 0
```

Keeping (-2 1) and hiding *,
Using lemma cv_base_result,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC4.2`.
Q.E.D.

C.11.27 ArithmeticExpressions.minus_ptr_ptr_TCC5

Terse proof for `minus_ptr_ptr_TCC5`.

`minus_ptr_ptr_TCC5`:

```
-------------------------------------------------------------------
{1}  ∀ (typ:
          Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                              (λ (p: (pointer?)):
                                  ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))),
        ptr_val1: (range_pointer(cv_base(typ))),
        ptr_val2: (range_pointer(cv_base(typ))):
       (array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ∧
        not_null?(typ)(ptr_val1) ∧
         not_null?(typ)(ptr_val2) ∧
          same_array(typ)(ptr_val1, ptr_val2) ∧
           up?(address_of(typ)(ptr_val1)) ∧ up?(address_of(typ)(ptr_val2))
       ⊃
       (∀ (difference:
             {q: int |
                   offset(down [Memory_Address](address_of(typ)(ptr_val1))) − offset(down
                    =
                    rem(size_of(typ(cv_base(typ))))(offset(down [Memory_Address](address
        difference =
         ndiv(offset(down(address_of(typ)(ptr_val1))) − offset(down(address_of(typ)(ptr_val2))),
              size_of(typ(cv_base(typ))))
        ⊃ Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int))
```

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`minus_ptr_ptr_TCC5.1:`

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC5.1`.
`minus_ptr_ptr_TCC5.2:`

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC5.2`.
Q.E.D.

C.11.28 ArithmeticExpressions.minus_ptr_ptr_TCC6

Terse proof for `minus_ptr_ptr_TCC6`.
`minus_ptr_ptr_TCC6:`

{1} ∀ (typ:
Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)):
¬ void?(typ(p)) ∧ ¬ function?(typ(p)))),
ptr_val1: (range_pointer(cv_base(typ))),
ptr_val2: (range_pointer(cv_base(typ)))):
¬ ((array?(typ(cv_base(typ))) ⊃ size_of(typ(cv_base(typ))) > 0) ∧
not_null?(typ)(ptr_val1) ∧
not_null?(typ)(ptr_val2) ∧
same_array(typ)(ptr_val1, ptr_val2) ∧
up?(address_of(typ)(ptr_val1)) ∧ up?(address_of(typ)(ptr_val2)))
⊃ Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)

Repeatedly Skolemizing and flattening,
Keeping (2) and hiding *,
Applying propositional simplification,
we get 2 subgoals:

`minus_ptr_ptr_TCC6.1`:

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC6.1`.
`minus_ptr_ptr_TCC6.2`:

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC6.2`.
Q.E.D.

C.11.29 ArithmeticExpressions.minus_ptr_ptr_TCC7

Terse proof for `minus_ptr_ptr_TCC7`.
`minus_ptr_ptr_TCC7`:

{1} ∀ (typ:
Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)):
¬ void?(typ(p)) ∧ ¬ function?(typ(p)))),
ptr_val1: (range_pointer(cv_base(typ)))):
pointer?(cv_base(typ))

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:
`minus_ptr_ptr_TCC7.1`:

{-1} Cpp_Type?(typ′)
{-2} cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
(λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
(typ′)
{-3} range_pointer(cv_base(typ′))(ptr_val1′)

{1} cv(pointer?)(typ′)
{2} pointer?(cv_base(typ′))

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC7.1`.

`minus_ptr_ptr_TCC7.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
             (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
        (typ′)
{-3}  range_pointer(cv_base(typ′))(ptr_val1′)
-------------------------------------------------------------------------
{1}   (pointer? ⊆ interpreted?)
{2}   pointer?(cv_base(typ′))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC7.2`.
Q.E.D.

C.11.30 ArithmeticExpressions.minus_ptr_ptr_TCC8

Terse proof for `minus_ptr_ptr_TCC8`.
`minus_ptr_ptr_TCC8:`

```
{1}   ∀ (typ:
            Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                 (λ (p: (pointer?)):
                                      ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))),
          ptr_val1: (range_pointer(cv_base(typ)))):
        Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)
```

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`minus_ptr_ptr_TCC8.1:`

```
{1}   Cpp_Type?(int)
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC8.1`.
`minus_ptr_ptr_TCC8.2:`

```
{1}   cv(non_bool_integral_enum?)(int)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `minus_ptr_ptr_TCC8.2`.
Q.E.D.

C.11.31 ArithmeticExpressions.minus_ptr_ptr_TCC9

Terse proof for `minus_ptr_ptr_TCC9`.
`minus_ptr_ptr_TCC9:`

```
{1}  ∀ (typ:
           Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))):
       pointer?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:
`minus_ptr_ptr_TCC9.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
        (typ′)
------------------------------------------------------------------------
{1}   cv(pointer?)(typ′)
{2}   pointer?(cv_base(typ′))
```

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC9.1`.
`minus_ptr_ptr_TCC9.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                (λ (p: (pointer?)): ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))
        (typ′)
------------------------------------------------------------------------
{1}   (pointer? ⊆ interpreted?)
{2}   pointer?(cv_base(typ′))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC9.2`.
Q.E.D.

C.11.32 ArithmeticExpressions.minus_ptr_ptr_TCC10

Terse proof for `minus_ptr_ptr_TCC10`.
`minus_ptr_ptr_TCC10:`

```
{1}  ∀ (typ:
           Cpp_Subtype(cv(extend[Cpp_Type_, (pointer?), bool, FALSE]
                                (λ (p: (pointer?)):
                                     ¬ void?(typ(p)) ∧ ¬ function?(typ(p))))):
       Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)
```

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`minus_ptr_ptr_TCC10.1:`

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC10.1`.
`minus_ptr_ptr_TCC10.2:`

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `minus_ptr_ptr_TCC10.2`.
Q.E.D.

C.11.33 ArithmeticExpressions.cmp_pointer_TCC1

Terse proof for `cmp_pointer_TCC1`.
`cmp_pointer_TCC1:`

{1} ∀ (p_typ: Cpp_Subtype(cv(pointer?)), ptr_val1: (range_pointer(cv_base(p_typ)))): pointer?(cv_base(p_typ))

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cmp_pointer_TCC1`.
Q.E.D.

C.11.34 ArithmeticExpressions.cmp_pointer_TCC2

Terse proof for `cmp_pointer_TCC2`.
`cmp_pointer_TCC2:`

{1} ∀ (p_typ: Cpp_Subtype(cv(pointer?))): pointer?(cv_base(p_typ))

Repeatedly Skolemizing and flattening,

Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cmp_pointer_TCC2`.
Q.E.D.

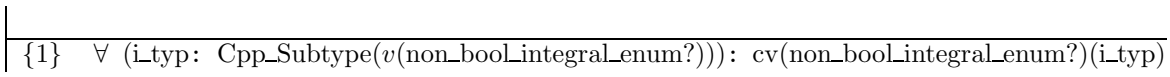
C.12 Proofs for AssemblerStatements (statements.pvs)

This theory contains no provable formal statements.

C.13 Proofs for AssignmentExpressions (expressions.pvs)

C.13.1 AssignmentExpressions.assign_times_TCC1

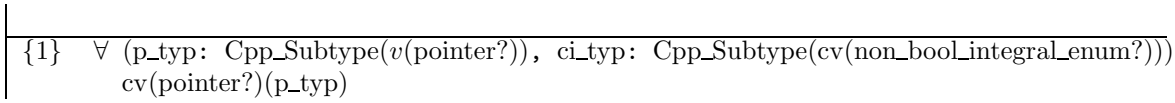
Terse proof for `assign_times_TCC1`.
`assign_times_TCC1:`



Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `assign_times_TCC1`.
Q.E.D.

C.13.2 AssignmentExpressions.assign_plus_ptr_TCC1

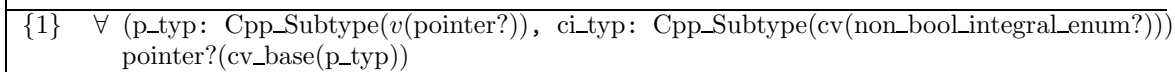
Terse proof for `assign_plus_ptr_TCC1`.
`assign_plus_ptr_TCC1:`



Repeatedly Skolemizing and flattening,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `assign_plus_ptr_TCC1`.
Q.E.D.

C.13.3 AssignmentExpressions.assign_plus_ptr_TCC2

Terse proof for `assign_plus_ptr_TCC2`.
`assign_plus_ptr_TCC2:`



Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,

we get 2 subgoals:
`assign_plus_ptr_TCC2.1:`

{-1}	Cpp_Type?(p_typ′)
{-2}	v(pointer?)(p_typ′)
{-3}	Cpp_Type?(ci_typ′)
{-4}	cv(non_bool_integral_enum?)(ci_typ′)
{1}	cv(pointer?)(p_typ′)
{2}	pointer?(cv_base(p_typ′))

Using lemma assign_plus_ptr_TCC1,
This completes the proof of `assign_plus_ptr_TCC2.1`.
`assign_plus_ptr_TCC2.2:`

{-1}	Cpp_Type?(p_typ′)
{-2}	v(pointer?)(p_typ′)
{-3}	Cpp_Type?(ci_typ′)
{-4}	cv(non_bool_integral_enum?)(ci_typ′)
{1}	(pointer? $\subseteq$ interpreted?)
{2}	pointer?(cv_base(p_typ′))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `assign_plus_ptr_TCC2.2`.
Q.E.D.

C.14 Proofs for AssignmentExpressions2 (expressions.pvs)

This theory contains no provable formal statements.

C.15 Proofs for Bitlist_ops (bits.pvs)

C.15.1 Bitlist_ops.binary_not_TCC1

Terse proof for `binary_not_TCC1`.
`binary_not_TCC1:`

{1}	$(\forall\ (x\colon \text{nat})\colon x < \text{expt}(2,\ \text{bits\_per\_byte}) \equiv x < \text{max\_byte}) \wedge$ $(\forall\ (x_1\colon \text{below}[\text{expt}(2,\ \text{bits\_per\_byte})])\colon \text{binary\_not}[\text{bits\_per\_byte}](x_1) < \text{max\_byte})$

Expanding the definition of max_byte,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_not_TCC1`.
Q.E.D.

C.15.2 Bitlist_ops.binary_and_TCC1

Terse proof for `binary_and_TCC1`.

`binary_and_TCC1:`

{1} $(\forall\ (x\colon\ \text{nat})\colon\ x < \text{expt}(2,\ \text{bits\_per\_byte}) \equiv x < \text{max\_byte}) \wedge$
$(\forall\ (x_1\colon\ [\text{below}[\text{expt}(2,\ \text{bits\_per\_byte})],\ \text{below}[\text{expt}(2,\ \text{bits\_per\_byte})]])\colon$
$\text{binary\_and}[\text{bits\_per\_byte}](x_1) < \text{max\_byte})$

Expanding the definition of max_byte,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_and_TCC1`.
Q.E.D.

C.15.3 Bitlist_ops.binary_or_TCC1

Terse proof for `binary_or_TCC1`.
`binary_or_TCC1:`

{1} $(\forall\ (x\colon\ \text{nat})\colon\ x < \text{expt}(2,\ \text{bits\_per\_byte}) \equiv x < \text{max\_byte}) \wedge$
$(\forall\ (x_1\colon\ [\text{below}[\text{expt}(2,\ \text{bits\_per\_byte})],\ \text{below}[\text{expt}(2,\ \text{bits\_per\_byte})]])\colon$
$\text{binary\_or}[\text{bits\_per\_byte}](x_1) < \text{max\_byte})$

Expanding the definition of max_byte,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_or_TCC1`.
Q.E.D.

C.15.4 Bitlist_ops.binary_xor_TCC1

Terse proof for `binary_xor_TCC1`.
`binary_xor_TCC1:`

{1} $(\forall\ (x\colon\ \text{nat})\colon\ x < \text{expt}(2,\ \text{bits\_per\_byte}) \equiv x < \text{max\_byte}) \wedge$
$(\forall\ (x_1\colon\ [\text{below}[\text{expt}(2,\ \text{bits\_per\_byte})],\ \text{below}[\text{expt}(2,\ \text{bits\_per\_byte})]])\colon$
$\text{binary\_xor}[\text{bits\_per\_byte}](x_1) < \text{max\_byte})$

Expanding the definition of max_byte,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_xor_TCC1`.
Q.E.D.

C.15.5 Bitlist_ops.singleton_zero_mask

Terse proof for `singleton_zero_mask`.
`singleton_zero_mask:`

{1} $\forall\ (s\colon\ \text{nat})\colon\ \text{singleton?}(\text{zero\_mask?}(s))$

Inducting on s on formula 1,
we get 2 subgoals:

`singleton_zero_mask.1:`

{1} singleton?(zero_mask?(0))

Expanding the definition of zero_mask?,
Expanding the definition of length,
Expanding the definition of singleton?,
Instantiating quantified variables,
we get 3 subgoals:
`singleton_zero_mask.1.1:`

{1} ∀ (y_1:
(λ (l: list[Byte]):
CASES l OF null: 0, cons(x, y): length(y) + 1 ENDCASES = 0 ∧
every(λ (t: Byte): t = 0)(l))):
null = y_1

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `singleton_zero_mask.1.1`.
`singleton_zero_mask.1.2:`

{1} every[Byte](λ (t: Byte): t = 0)(null[Byte])

Expanding the definition of every,
which is trivially true.
This completes the proof of `singleton_zero_mask.1.2`.
`singleton_zero_mask.1.3:`

{1} every[Byte](λ (t: Byte): t = 0)(null[Byte])
{2} ∃ (x_1:
(λ (l: list[Byte]):
CASES l OF null: 0, cons(x, y): length(y) + 1 ENDCASES = 0 ∧
every(λ (t: Byte): t = 0)(l))):
∀ (y_1:
(λ (l: list[Byte]):
CASES l OF null: 0, cons(x, y): length(y) + 1 ENDCASES = 0 ∧
every(λ (t: Byte): t = 0)(l))):
x_1 = y_1

Expanding the definition of every,
which is trivially true.
This completes the proof of `singleton_zero_mask.1.3`.
`singleton_zero_mask.2:`

{1} ∀ j: singleton?(zero_mask?(j)) ⊃ singleton?(zero_mask?(j + 1))

Repeatedly Skolemizing and flattening,
Expanding the definition of zero_mask?,
Expanding the definition of singleton?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (cons(0, x!1)),

we get 3 subgoals:

`singleton_zero_mask.2.1:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (x′)
{-2}  length(x′) = j′
{-3}  every(λ (t: Byte): t = 0)(x′)
{-4}  j′ ≥ 0
{-5}  ∀ (y:
           (λ (l: list[Byte]):
              length(l) = j′ ∧ every(λ (t: Byte): t = 0)(l))):
        x′ = y
------------------------------------------------------------------------
{1}   ∀ (y:
           (λ (l: list[Byte]):
              length(l) = 1 + j′ ∧ every(λ (t: Byte): t = 0)(l))):
        cons(0, x′) = y
```

Repeatedly Skolemizing and flattening,

Case splitting on null?(y!1),

we get 2 subgoals:

`singleton_zero_mask.2.1.1:`

```
{-1}  null?(y′)
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (y′)
{-3}  length(y′) = 1 + j′
{-4}  every(λ (t: Byte): t = 0)(y′)
{-5}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (x′)
{-6}  length(x′) = j′
{-7}  every(λ (t: Byte): t = 0)(x′)
{-8}  j′ ≥ 0
{-9}  ∀ (y:
           (λ (l: list[Byte]):
              length(l) = j′ ∧ every(λ (t: Byte): t = 0)(l))):
        x′ = y
------------------------------------------------------------------------
{1}   cons(0, x′) = y′
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `singleton_zero_mask.2.1.1`.

`singleton_zero_mask.2.1.2:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (y′)
{-2}  length(y′) = 1 + j′
{-3}  every(λ (t: Byte): t = 0)(y′)
{-4}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (x′)
{-5}  length(x′) = j′
{-6}  every(λ (t: Byte): t = 0)(x′)
{-7}  j′ ≥ 0
{-8}  ∀ (y:
            (λ (l: list[Byte]):
               length(l) = j′ ∧ every(λ (t: Byte): t = 0)(l))):
        x′ = y
|-----
{1}   null?(y′)
{2}   cons(0, x′) = y′
```

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in -8 with the terms: (cdr(y!1)),
we get 2 subgoals:
`singleton_zero_mask.2.1.2.1:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (y′)
{-2}  length(cdr(y′)) = j′
{-3}  every(λ (t: Byte): t = 0)(y′)
{-4}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (x′)
{-5}  CASES x′ OF null: 0, cons(x, y): length(y) + 1 ENDCASES = j′
{-6}  every(λ (t: Byte): t = 0)(x′)
{-7}  j′ ≥ 0
{-8}  x′ = cdr(y′)
|-----
{1}   null?(y′)
{2}   cons(0, x′) = y′
```

Expanding the definition of every,
Replacing using formula -8,
Using lemma list_cons_eta,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `singleton_zero_mask.2.1.2.1`.

`singleton_zero_mask.2.1.2.2:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (y′)
{-2}  length(cdr(y′)) = j′
{-3}  every(λ (t: Byte): t = 0)(y′)
{-4}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (x′)
{-5}  CASES x′ OF null: 0, cons(x, y): length(y) + 1 ENDCASES = j′
{-6}  every(λ (t: Byte): t = 0)(x′)
{-7}  j′ ≥ 0
------------------------------------------------------------------------
{1}   every[Byte](λ (t: Byte): t = 0)(cdr[Byte](y′))
{2}   null?(y′)
{3}   cons(0, x′) = y′
```

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `singleton_zero_mask.2.1.2.2`.

`singleton_zero_mask.2.2:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (x′)
{-2}  length(x′) = j′
{-3}  every(λ (t: Byte): t = 0)(x′)
{-4}  j′ ≥ 0
{-5}  ∀ (y:
           (λ (l: list[Byte]):
                length(l) = j′ ∧ every(λ (t: Byte): t = 0)(l))):
        x′ = y
------------------------------------------------------------------------
{1}   length[Byte](cons[Byte](0, x′)) = 1 + j′ ∧
       every[Byte](λ (t: Byte): t = 0)(cons[Byte](0, x′))
```

Expanding the definition of length,

Expanding the definition of every,

which is trivially true.

This completes the proof of `singleton_zero_mask.2.2`.

singleton_zero_mask.2.3:

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
        (x′)
{-2}  length(x′) = j′
{-3}  every(λ (t: Byte): t = 0)(x′)
{-4}  j′ ≥ 0
{-5}  ∀ (y:
           (λ (l: list[Byte]):
              length(l) = j′ ∧ every(λ (t: Byte): t = 0)(l))):
        x′ = y
|-------
{1}   0 < max_byte
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of singleton_zero_mask.2.3.
Q.E.D.

C.15.6 Bitlist_ops.bit_set?_TCC1

Terse proof for bit_set?_TCC1.

bit_set?_TCC1:

```
|-------
{1}   ∀ (n: nat, l: list[Byte]): n < length(l) × max_byte ⊃ max_byte > 0
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of bit_set?_TCC1.
Q.E.D.

C.15.7 Bitlist_ops.bit_set?_TCC2

Terse proof for bit_set?_TCC2.

bit_set?_TCC2:

```
|-------
{1}   ∀ (n: nat, l: list[Byte]):
        n < length(l) × max_byte ⊃ ndiv(n, max_byte) < length[Byte](l)
```

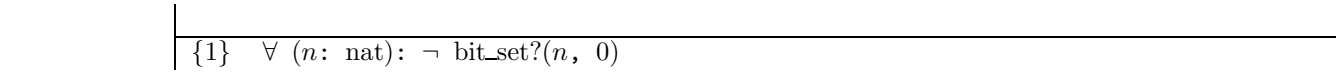
Repeatedly Skolemizing and flattening,
Using lemma ndiv_lt,
Using lemma both_sides_div_pos_lt1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of bit_set?_TCC2.
Q.E.D.

C.16 Proofs for Bitops (bits.pvs)

C.16.1 Bitops.bit_set_zero_mask

Terse proof for bit_set_zero_mask.

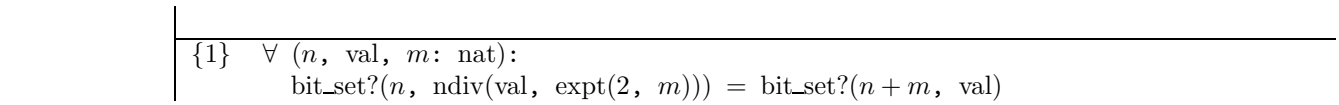
`bit_set_zero_mask:`



Expanding the definition of bit_set?,
Repeatedly Skolemizing and flattening,
Rewriting using cut_bit_zero, matching in *,
This completes the proof of `bit_set_zero_mask`.
Q.E.D.

C.16.2 Bitops.bit_set_ndiv_expt2

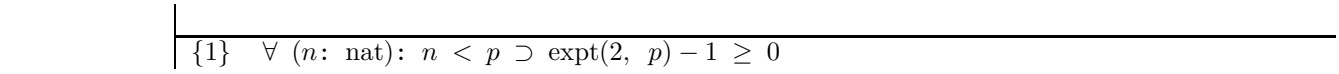
Terse proof for `bit_set_ndiv_expt2`.
`bit_set_ndiv_expt2:`



Repeatedly Skolemizing and flattening,
Expanding the definition of bit_set?,
Expanding the definition of cut_bit,
Using lemma ndiv_times_2,
Rewriting using expt_plus_aux, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `bit_set_ndiv_expt2`.
Q.E.D.

C.16.3 Bitops.bit_set_full_mask_TCC1

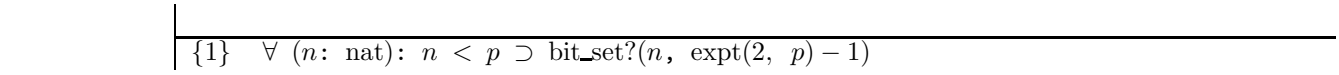
Terse proof for `bit_set_full_mask_TCC1`.
`bit_set_full_mask_TCC1:`



Repeatedly Skolemizing and flattening,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bit_set_full_mask_TCC1`.
Q.E.D.

C.16.4 Bitops.bit_set_full_mask

Terse proof for `bit_set_full_mask`.
`bit_set_full_mask:`



Repeatedly Skolemizing and flattening,
Expanding the definition of bit_set?,
Expanding the definition of cut_bit,
Using lemma ndiv_plus_1,

Case splitting on divides(expt(2, n!1), expt(2, p)),

we get 2 subgoals:

`bit_set_full_mask.1:`

$$
\begin{array}{ll}
\{\text{-1}\} & \text{divides}(\text{expt}(2,\ n'),\ \text{expt}(2,\ p)) \\
\{\text{-2}\} & \text{divides}(\text{expt}(2,\ n'),\ \text{expt}(2,\ p)) \supset \\
 & \quad \text{ndiv}(\text{expt}(2,\ p) + -1,\ \text{expt}(2,\ n')) = \\
 & \quad\quad \text{ndiv}(\text{expt}(2,\ p),\ \text{expt}(2,\ n')) + \text{ndiv}(-1,\ \text{expt}(2,\ n')) \\
\{\text{-3}\} & n' \geq 0 \\
\{\text{-4}\} & n' < p \\
\hline
\{1\} & \text{rem}(2)(\text{ndiv}(\text{expt}(2,\ p) - 1,\ \text{expt}(2,\ n'))) = 1
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -2,

Rewriting using ndiv_expt_expt, matching in *,

Using lemma rem_sum1,

Replacing using formula -1,

Expanding the definition of expt,

Rewriting using rem_multiple1, matching in *,

Simplifying, rewriting, and recording with decision procedures,

Using lemma rem_def,

Using lemma ndiv_plus_2,

Rewriting using divides_reflexive, matching in *,

Rewriting using ndiv_self, matching in *,

Rewriting using ndiv_bigger, matching in *,

we get 2 subgoals:

`bit_set_full_mask.1.1:`

$$
\begin{array}{ll}
\{\text{-1}\} & 0 = \text{ndiv}(-1,\ \text{expt}(2,\ n')) + 1 \\
\{\text{-2}\} & \text{rem}(2)(\text{ndiv}(-1,\ \text{expt}(2,\ n'))) = 1 \equiv \\
 & \quad (\exists\ q\text{:}\ \text{ndiv}(-1,\ \text{expt}(2,\ n')) = 2 \times q + 1) \\
\{\text{-3}\} & \text{rem}(2)(\text{rem}(2)(\text{expt}(2,\ p - n')) + \text{ndiv}(-1,\ \text{expt}(2,\ n'))) = \\
 & \quad \text{rem}(2)(\text{ndiv}(-1,\ \text{expt}(2,\ n')) + \text{expt}(2,\ p - n')) \\
\{\text{-4}\} & \text{divides}(\text{expt}(2,\ n'),\ \text{expt}(2,\ p)) \\
\{\text{-5}\} & \text{ndiv}(\text{expt}(2,\ p) - 1,\ \text{expt}(2,\ n')) = \\
 & \quad \text{ndiv}(-1,\ \text{expt}(2,\ n')) + \text{expt}(2,\ p - n') \\
\{\text{-6}\} & n' \geq 0 \\
\{\text{-7}\} & n' < p \\
\hline
\{1\} & \text{rem}(2)(\text{ndiv}(-1,\ \text{expt}(2,\ n'))) = 1
\end{array}
$$

Using lemma both_sides_plus1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Instantiating the top quantifier in 2 with the terms: (-1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.1.1`.

`bit_set_full_mask.1.2:`

{-1}	$\mathrm{ndiv}(\mathrm{expt}(2,\ n') - 1,\ \mathrm{expt}(2,\ n')) = 1 + \mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n'))$
{-2}	$\mathrm{rem}(2)(\mathrm{rem}(2)(\mathrm{expt}(2,\ p - n')) + \mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n'))) =$ $\mathrm{rem}(2)(\mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n')) + \mathrm{expt}(2,\ p - n'))$
{-3}	$\mathrm{divides}(\mathrm{expt}(2,\ n'),\ \mathrm{expt}(2,\ p))$
{-4}	$\mathrm{ndiv}(\mathrm{expt}(2,\ p) - 1,\ \mathrm{expt}(2,\ n')) =$ $\mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n')) + \mathrm{expt}(2,\ p - n')$
{-5}	$n' \geq 0$
{-6}	$n' < p$
{1}	$\mathrm{expt}(2,\ n') - 1 \geq 0$
{2}	$\exists\ q\colon\ \mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n')) = 1 + 2 \times q$
{3}	$\mathrm{rem}(2)(\mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n'))) = 1$

Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bit_set_full_mask.1.2`.
`bit_set_full_mask.2:`

{-1}	$\mathrm{divides}(\mathrm{expt}(2,\ n'),\ \mathrm{expt}(2,\ p)) \supset$ $\mathrm{ndiv}(\mathrm{expt}(2,\ p) + -1,\ \mathrm{expt}(2,\ n')) =$ $\mathrm{ndiv}(\mathrm{expt}(2,\ p),\ \mathrm{expt}(2,\ n')) + \mathrm{ndiv}(-1,\ \mathrm{expt}(2,\ n'))$
{-2}	$n' \geq 0$
{-3}	$n' < p$
{1}	$\mathrm{divides}(\mathrm{expt}(2,\ n'),\ \mathrm{expt}(2,\ p))$
{2}	$\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2,\ p) - 1,\ \mathrm{expt}(2,\ n'))) = 1$

Hiding formulas: (-1 2),
Expanding the definition of divides,
Instantiating the top quantifier in 1 with the terms: (expt(2, p - n!1)),
we get 2 subgoals:
`bit_set_full_mask.2.1:`

{-1}	$n' \geq 0$
{-2}	$n' < p$
{1}	$\mathrm{expt}(2,\ p) = \mathrm{expt}(2,\ n') \times \mathrm{expt}(2,\ p - n')$

Using lemma expt_plus_aux,
we get 2 subgoals:
`bit_set_full_mask.2.1.1:`

{-1}	$\mathrm{expt}(2,\ n' + p - n') = \mathrm{expt}(2,\ n') \times \mathrm{expt}(2,\ p - n')$
{-2}	$n' \geq 0$
{-3}	$n' < p$
{1}	$\mathrm{expt}(2,\ p) = \mathrm{expt}(2,\ n') \times \mathrm{expt}(2,\ p - n')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bit_set_full_mask.2.1.1`.
`bit_set_full_mask.2.1.2:`

{-1}	$n' \geq 0$
{-2}	$n' < p$
{1}	$p - n' \geq 0$
{2}	$\mathrm{expt}(2,\ p) = \mathrm{expt}(2,\ n') \times \mathrm{expt}(2,\ p - n')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.2.1.2`.

`bit_set_full_mask.2.2`:

$$\begin{array}{ll} \{\text{-1}\} & n' \geq 0 \\ \{\text{-2}\} & n' < p \\ \hline \{1\} & p - n' \geq 0 \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bit_set_full_mask.2.2`.

Q.E.D.

C.16.5 Bitops.bits_equal

Terse proof for `bits_equal`.

`bits_equal`:

$$\begin{array}{ll} \{1\} & \forall\ (v_1,\ v_2,\ p\colon \text{nat})\colon \\ & \quad (\forall\ (n\colon \text{nat})\colon n < p \supset \text{bit\_set?}(n,\ v_1) = \text{bit\_set?}(n,\ v_2)) \supset \\ & \quad\ \text{rem}(\text{expt}(2,\ p))(v_1) = \text{rem}(\text{expt}(2,\ p))(v_2) \end{array}$$

Inducting on p on formula 1,

we get 2 subgoals:

`bits_equal.1`:

$$\begin{array}{ll} \{1\} & \forall\ (v_1,\ v_2\colon \text{nat})\colon \\ & \quad (\forall\ (n\colon \text{nat})\colon n < 0 \supset \text{bit\_set?}(n,\ v_1) = \text{bit\_set?}(n,\ v_2)) \supset \\ & \quad\ \text{rem}(\text{expt}(2,\ 0))(v_1) = \text{rem}(\text{expt}(2,\ 0))(v_2) \end{array}$$

Repeatedly Skolemizing and flattening,

Expanding the definition of expt,

Rewriting using rem_def, matching in *,

Instantiating the top quantifier in 1 with the terms: (v1!1 - rem(1)(v2!1)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal.1`.

`bits_equal.2`:

$$\begin{array}{ll} \{1\} & \forall\ j\colon \\ & \quad (\forall\ (v_1,\ v_2\colon \text{nat})\colon \\ & \qquad (\forall\ (n\colon \text{nat})\colon n < j \supset \text{bit\_set?}(n,\ v_1) = \text{bit\_set?}(n,\ v_2)) \supset \\ & \qquad\ \text{rem}(\text{expt}(2,\ j))(v_1) = \text{rem}(\text{expt}(2,\ j))(v_2)) \\ & \quad \supset \\ & \quad (\forall\ (v_1,\ v_2\colon \text{nat})\colon \\ & \qquad (\forall\ (n\colon \text{nat})\colon n < j+1 \supset \text{bit\_set?}(n,\ v_1) = \text{bit\_set?}(n,\ v_2)) \supset \\ & \qquad\ \text{rem}(\text{expt}(2,\ j+1))(v_1) = \text{rem}(\text{expt}(2,\ j+1))(v_2)) \end{array}$$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -4 with the terms: (v1!1 v2!1),

Splitting conjunctions,

we get 2 subgoals:

`bits_equal.2.1:`

{-1}	$\text{rem}(\text{expt}(2,\ j'))(v_1') = \text{rem}(\text{expt}(2,\ j'))(v_2')$
{-2}	$v_1' \geq 0$
{-3}	$v_2' \geq 0$
{-4}	$j' \geq 0$
{-5}	$\forall\ (n\text{: nat}):\ n < j'+1 \supset \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$
{1}	$\text{rem}(\text{expt}(2,\ j'+1))(v_1') = \text{rem}(\text{expt}(2,\ j'+1))(v_2')$

Expanding the definition of expt,
Expanding the definition of bit_set?,
Expanding the definition of cut_bit,
Using lemma rem_prod_3,
Using lemma rem_prod_3,
Replacing using formula -1,
Replacing using formula -2,
Replacing using formula -3,
Instantiating the top quantifier in -7 with the terms: (j!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal.2.1`.
`bits_equal.2.2:`

{-1}	$v_1' \geq 0$
{-2}	$v_2' \geq 0$
{-3}	$j' \geq 0$
{-4}	$\forall\ (n\text{: nat}):\ n < j'+1 \supset \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$
{1}	$\forall\ (n\text{: nat}):\ n < j' \supset \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$
{2}	$\text{rem}(\text{expt}(2,\ j'+1))(v_1') = \text{rem}(\text{expt}(2,\ j'+1))(v_2')$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -6 with the terms: (n!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal.2.2`.
Q.E.D.

C.16.6 Bitops.bits_equal2

Terse proof for `bits_equal2`.
`bits_equal2:`

{1}	$\forall\ (v_1,\ v_2\text{: below}\lfloor\text{expt}(2,\ p)\rfloor):$ $(\forall\ (n\text{: nat}):\ n < p \supset \text{bit\_set?}(n,\ v_1) = \text{bit\_set?}(n,\ v_2)) \equiv v_1 = v_2$

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
`bits_equal2.1:`

{-1}	$\forall\ (n\text{: nat}):\ n < p \supset \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$
{-2}	$v_1' < \text{expt}(2,\ p)$
{-3}	$v_2' < \text{expt}(2,\ p)$
{1}	$v_1' = v_2'$

Using lemma bits_equal,
Replacing using formula -2,

Rewriting using rem_mod, matching in *,

Rewriting using rem_mod, matching in *,

This completes the proof of `bits_equal2.1`.

`bits_equal2.2`:

$$\begin{array}{ll}
\{\text{-1}\} & v_1' = v_2' \\
\{\text{-2}\} & v_1' < \text{expt}(2, p) \\
\{\text{-3}\} & v_2' < \text{expt}(2, p) \\
\hline
\{1\} & \forall\ (n\text{: nat})\text{: } n < p \supset \text{bit_set?}(n, v_1') = \text{bit_set?}(n, v_2')
\end{array}$$

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal2.2`.

Q.E.D.

C.16.7 Bitops.bits_equal3

Terse proof for `bits_equal3`.

`bits_equal3`:

$$\begin{array}{ll}
 & \\
\hline
\{1\} & \forall\ (v_1, v_2\text{: nat})\text{: } (\forall\ (n\text{: nat})\text{: } \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset v_1 = v_2
\end{array}$$

Repeatedly Skolemizing and flattening,

Applying bits_equal

Case splitting on EXISTS (x: posnat): v1!1 < expt(2, x) AND v2!1 < expt(2, x),

we get 2 subgoals:

`bits_equal3.1`:

$$\begin{array}{ll}
\{\text{-1}\} & \exists\ (x\text{: posnat})\text{: } v_1' < \text{expt}(2, x) \wedge v_2' < \text{expt}(2, x) \\
\{\text{-2}\} & \forall\ (v_1, v_2, p\text{: nat})\text{:} \\
 & \quad (\forall\ (n\text{: nat})\text{: } n < p \supset \text{bit_set?}(n, v_1) = \text{bit_set?}(n, v_2)) \supset \\
 & \quad \text{rem}(\text{expt}(2, p))(v_1) = \text{rem}(\text{expt}(2, p))(v_2) \\
\{\text{-3}\} & v_1' \geq 0 \\
\{\text{-4}\} & v_2' \geq 0 \\
\{\text{-5}\} & \forall\ (n\text{: nat})\text{: } \text{bit_set?}(n, v_1') = \text{bit_set?}(n, v_2') \\
\hline
\{1\} & v_1' = v_2'
\end{array}$$

Instantiating the top quantifier in -2 with the terms: (v1!1 v2!1 choose({p: posnat | v1!1 < expt(2, p) AND v2!1 < expt(2, p)})),

we get 2 subgoals:

`bits_equal3.1.1:`

{-1} $\exists\ (x\colon \text{posnat})\colon v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{-2} $(\forall\ (n\colon \text{nat})\colon$
$n < \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\}) \supset$
$\text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2'))$
$\supset$
$\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_1')
$=$
$\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_2')
{-3} $v_1' \geq 0$
{-4} $v_2' \geq 0$
{-5} $\forall\ (n\colon \text{nat})\colon \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$

{1} $v_1' = v_2'$

Splitting conjunctions,
we get 2 subgoals:
`bits_equal3.1.1.1:`

{-1} $\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_1')
$=$
$\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_2')
{-2} $\exists\ (x\colon \text{posnat})\colon v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{-3} $v_1' \geq 0$
{-4} $v_2' \geq 0$
{-5} $\forall\ (n\colon \text{nat})\colon \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$

{1} $v_1' = v_2'$

Using lemma rem_mod,
we get 3 subgoals:
`bits_equal3.1.1.1.1:`

{-1} $\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_1')
$= v_1'$
{-2} $\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_1')
$=$
$\text{rem}(\text{expt}(2,\ \text{choose}(\{p\colon \text{posnat} \mid v_1' < \text{expt}(2,\ p) \wedge v_2' < \text{expt}(2,\ p)\})))$
(v_2')
{-3} $\exists\ (x\colon \text{posnat})\colon v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{-4} $v_1' \geq 0$
{-5} $v_2' \geq 0$
{-6} $\forall\ (n\colon \text{nat})\colon \text{bit\_set?}(n,\ v_1') = \text{bit\_set?}(n,\ v_2')$

{1} $v_1' = v_2'$

Replacing using formula -1,
Hiding formulas: -1,
Using lemma rem_mod,
we get 3 subgoals:

`bits_equal3.1.1.1.1.1:`

```
{-1}  rem(expt(2, choose({p: posnat | v'1 < expt(2, p) ∧ v'2 < expt(2, p)})))
        (v'2)
      = v'2
{-2}  v'1 =
      rem(expt(2, choose({p: posnat | v'1 < expt(2, p) ∧ v'2 < expt(2, p)})))
        (v'2)
{-3}  ∃ (x: posnat): v'1 < expt(2, x) ∧ v'2 < expt(2, x)
{-4}  v'1 ≥ 0
{-5}  v'2 ≥ 0
{-6}  ∀ (n: nat): bit_set?(n, v'1) = bit_set?(n, v'2)
------------------------------------------------------------------------
{1}   v'1 = v'2
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal3.1.1.1.1.1`.

`bits_equal3.1.1.1.1.2:`

```
{-1}  v'1 =
      rem(expt(2, choose({p: posnat | v'1 < expt(2, p) ∧ v'2 < expt(2, p)})))
        (v'2)
{-2}  ∃ (x: posnat): v'1 < expt(2, x) ∧ v'2 < expt(2, x)
{-3}  v'1 ≥ 0
{-4}  v'2 ≥ 0
{-5}  ∀ (n: nat): bit_set?(n, v'1) = bit_set?(n, v'2)
------------------------------------------------------------------------
{1}   v'2 <
      expt(2, choose[posnat]({p: posnat | v'1 < expt(2, p) ∧ v'2 < expt(2, p)}))
{2}   v'1 = v'2
```

Expanding the definition of choose,

Using lemma epsilon_ax,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `bits_equal3.1.1.1.1.2`.

`bits_equal3.1.1.1.1.3:`

```
{-1}  v'1 =
      rem(expt(2, choose({p: posnat | v'1 < expt(2, p) ∧ v'2 < expt(2, p)})))
        (v'2)
{-2}  ∃ (x: posnat): v'1 < expt(2, x) ∧ v'2 < expt(2, x)
{-3}  v'1 ≥ 0
{-4}  v'2 ≥ 0
{-5}  ∀ (n: nat): bit_set?(n, v'1) = bit_set?(n, v'2)
------------------------------------------------------------------------
{1}   nonempty?[posnat]({p: posnat | v'1 < expt(2, p) ∧ v'2 < expt(2, p)})
{2}   v'1 = v'2
```

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Rewriting using forall_not, matching in *,

Expanding the definition of member,

which is trivially true.

This completes the proof of `bits_equal3.1.1.1.1.3`.

`bits_equal3.1.1.1.2:`

{-1} rem(expt(2, choose({p: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$})) ($v_1'$) = rem(expt(2, choose({$p$: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$})) ($v_2'$)
{-2} $\exists$ (x: posnat): $v_1' < \text{expt}(2, x) \wedge v_2' < \text{expt}(2, x)$
{-3} $v_1' \geq 0$
{-4} $v_2' \geq 0$
{-5} $\forall$ (n: nat): bit_set?(n, v_1') = bit_set?(n, v_2')

{1} $v_1' <$ expt(2, choose⌊posnat⌋({p: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$}))
{2} $v_1' = v_2'$

Expanding the definition of choose,
Using lemma epsilon_ax,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.1.1.1.2`.

`bits_equal3.1.1.1.3:`

{-1} rem(expt(2, choose({p: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$})) ($v_1'$) = rem(expt(2, choose({$p$: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$})) ($v_2'$)
{-2} $\exists$ (x: posnat): $v_1' < \text{expt}(2, x) \wedge v_2' < \text{expt}(2, x)$
{-3} $v_1' \geq 0$
{-4} $v_2' \geq 0$
{-5} $\forall$ (n: nat): bit_set?(n, v_1') = bit_set?(n, v_2')

{1} nonempty?⌊posnat⌋({p: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$})
{2} $v_1' = v_2'$

Expanding the definition of nonempty?,
Expanding the definition of empty?,
Rewriting using forall_not, matching in *,
Expanding the definition of member,
which is trivially true.
This completes the proof of `bits_equal3.1.1.1.3`.

`bits_equal3.1.1.2:`

{-1} $\exists$ (x: posnat): $v_1' < \text{expt}(2, x) \wedge v_2' < \text{expt}(2, x)$
{-2} $v_1' \geq 0$
{-3} $v_2' \geq 0$
{-4} $\forall$ (n: nat): bit_set?(n, v_1') = bit_set?(n, v_2')

{1} $\forall$ (n: nat): $n <$ choose({p: posnat | $v_1' < \text{expt}(2, p) \wedge v_2' < \text{expt}(2, p)$}) $\supset$ bit_set?(n, v_1') = bit_set?(n, v_2')
{2} $v_1' = v_2'$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `bits_equal3.1.1.2`.

`bits_equal3.1.2:`

```
{-1}  ∃ (x: posnat): v1' < expt(2, x) ∧ v2' < expt(2, x)
{-2}  v1' ≥ 0
{-3}  v2' ≥ 0
{-4}  ∀ (n: nat): bit_set?(n, v1') = bit_set?(n, v2')
|-------
{1}   nonempty?[posnat]({p: posnat | v1' < expt(2, p) ∧ v2' < expt(2, p)})
{2}   v1' = v2'
```

Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
Rewriting using forall_not, matching in *,
This completes the proof of `bits_equal3.1.2`.
`bits_equal3.2:`

```
{-1}  ∀ (v1, v2, p: nat):
        (∀ (n: nat): n < p ⊃ bit_set?(n, v1) = bit_set?(n, v2)) ⊃
         rem(expt(2, p))(v1) = rem(expt(2, p))(v2)
{-2}  v1' ≥ 0
{-3}  v2' ≥ 0
{-4}  ∀ (n: nat): bit_set?(n, v1') = bit_set?(n, v2')
|-------
{1}   ∃ (x: posnat): v1' < expt(2, x) ∧ v2' < expt(2, x)
{2}   v1' = v2'
```

Using lemma large_expt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Copying formula number: -1
Hiding formulas: (-3 -6),
Case splitting on v1!1 /= 0 AND v2!1 /= 0,
we get 2 subgoals:
`bits_equal3.2.1:`

```
{-1}  v1' ≠ 0 ∧ v2' ≠ 0
{-2}  ∀ py: ∃ n: py < expt(2, n)
{-3}  ∀ py: ∃ n: py < expt(2, n)
{-4}  v1' ≥ 0
{-5}  v2' ≥ 0
|-------
{1}   ∃ (x: posnat): v1' < expt(2, x) ∧ v2' < expt(2, x)
{2}   v1' = v2'
```

Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -1 with the terms: (v1!1),
we get 2 subgoals:
`bits_equal3.2.1.1:`

```
{-1}  ∃ n: v1' < expt(2, n)
{-2}  ∀ py: ∃ n: py < expt(2, n)
{-3}  v1' ≥ 0
{-4}  v2' ≥ 0
|-------
{1}   v1' = 0
{2}   v2' = 0
{3}   ∃ (x: posnat): v1' < expt(2, x) ∧ v2' < expt(2, x)
{4}   v1' = v2'
```

Instantiating the top quantifier in -2 with the terms: (v2!1),

we get 2 subgoals:
`bits_equal3.2.1.1.1:`

{-1}	$\exists\ n\colon\ v_1' < \mathrm{expt}(2,\ n)$
{-2}	$\exists\ n\colon\ v_2' < \mathrm{expt}(2,\ n)$
{-3}	$v_1' \geq 0$
{-4}	$v_2' \geq 0$
{1}	$v_1' = 0$
{2}	$v_2' = 0$
{3}	$\exists\ (x\colon\ \mathrm{posnat})\colon\ v_1' < \mathrm{expt}(2,\ x) \wedge v_2' < \mathrm{expt}(2,\ x)$
{4}	$v_1' = v_2'$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 3 with the terms: (n!1 + n!2 + 1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`bits_equal3.2.1.1.1.1:`

{-1}	$n'' \geq 0$
{-2}	$n' \geq 0$
{-3}	$v_1' < \mathrm{expt}(2,\ n')$
{-4}	$v_2' < \mathrm{expt}(2,\ n'')$
{-5}	$v_1' \geq 0$
{-6}	$v_2' \geq 0$
{1}	$v_1' = 0$
{2}	$v_2' = 0$
{3}	$v_2' < \mathrm{expt}(2,\ 1 + n' + n'')$
{4}	$v_1' = v_2'$

Rewriting using expt_plus_aux, matching in *,
Using lemma lt_times_lt_pos2,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.2.1.1.1.1`.
`bits_equal3.2.1.1.1.2:`

{-1}	$n'' \geq 0$
{-2}	$n' \geq 0$
{-3}	$v_1' < \mathrm{expt}(2,\ n')$
{-4}	$v_2' < \mathrm{expt}(2,\ n'')$
{-5}	$v_1' \geq 0$
{-6}	$v_2' \geq 0$
{1}	$v_1' = 0$
{2}	$v_2' = 0$
{3}	$v_1' < \mathrm{expt}(2,\ 1 + n' + n'')$
{4}	$v_1' = v_2'$

Rewriting using expt_plus_aux, matching in * where m gets n!1, n gets 1 + n!2,
Using lemma lt_times_lt_pos2,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.2.1.1.1.2`.

bits_equal3.2.1.1.2:

{-1}	$\exists\ n\colon\ v_1' < \text{expt}(2,\ n)$
{-2}	$v_1' \geq 0$
{-3}	$v_2' \geq 0$
{1}	$v_2' > 0$
{2}	$v_1' = 0$
{3}	$v_2' = 0$
{4}	$\exists\ (x\colon \text{posnat})\colon\ v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{5}	$v_1' = v_2'$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of bits_equal3.2.1.1.2.
bits_equal3.2.1.2:

{-1}	$\forall\ \text{py}\colon\ \exists\ n\colon\ \text{py} < \text{expt}(2,\ n)$
{-2}	$v_1' \geq 0$
{-3}	$v_2' \geq 0$
{1}	$v_1' > 0$
{2}	$v_1' = 0$
{3}	$v_2' = 0$
{4}	$\exists\ (x\colon \text{posnat})\colon\ v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{5}	$v_1' = v_2'$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of bits_equal3.2.1.2.
bits_equal3.2.2:

{-1}	$\forall\ \text{py}\colon\ \exists\ n\colon\ \text{py} < \text{expt}(2,\ n)$
{-2}	$\forall\ \text{py}\colon\ \exists\ n\colon\ \text{py} < \text{expt}(2,\ n)$
{-3}	$v_1' \geq 0$
{-4}	$v_2' \geq 0$
{1}	$v_1' \neq 0 \wedge v_2' \neq 0$
{2}	$\exists\ (x\colon \text{posnat})\colon\ v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{3}	$v_1' = v_2'$

Expanding the definition of /=,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
bits_equal3.2.2.1:

{-1}	$(v_1' = 0)$
{-2}	$\forall\ \text{py}\colon\ \exists\ n\colon\ \text{py} < \text{expt}(2,\ n)$
{-3}	$v_1' \geq 0$
{-4}	$v_2' \geq 0$
{1}	$\exists\ (x\colon \text{posnat})\colon\ v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x)$
{2}	$v_1' = v_2'$

Replacing using formula -1,
Instantiating the top quantifier in -2 with the terms: (v2!1),
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (n!1 + 1),
Rewriting using expt_plus_aux, matching in *,
Using lemma lt_times_lt_pos2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of expt,

Expanding the definition of expt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.2.2.1`.
`bits_equal3.2.2.2`:

$$
\begin{array}{ll}
\{\text{-}1\} & (v_2' = 0) \\
\{\text{-}2\} & \forall\ \text{py}\colon\ \exists\ n\colon\ \text{py} < \text{expt}(2,\ n) \\
\{\text{-}3\} & v_1' \geq 0 \\
\{\text{-}4\} & v_2' \geq 0 \\
\hline
\{1\} & \exists\ (x\colon\ \text{posnat})\colon\ v_1' < \text{expt}(2,\ x) \wedge v_2' < \text{expt}(2,\ x) \\
\{2\} & v_1' = v_2'
\end{array}
$$

Replacing using formula -1,
Instantiating the top quantifier in -2 with the terms: (v1!1),
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (n!1 + 1),
Rewriting using expt_plus_aux, matching in *,
Using lemma lt_times_lt_pos2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of expt,
Expanding the definition of expt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bits_equal3.2.2.2`.
Q.E.D.

C.16.8 Bitops.bits_equal4

Terse proof for `bits_equal4`.
`bits_equal4`:

$$
\begin{array}{ll}
\{1\} & \forall\ (v_1,\ v_2\colon\ \text{nat},\ m\colon\ \text{nat})\colon \\
& \quad (\forall\ (n\colon\ \text{nat})\colon\ n \geq m \supset \text{bit\_set?}(n,\ v_1) = \text{bit\_set?}(n,\ v_2)) \supset \\
& \quad\ \text{ndiv}(v_1,\ \text{expt}(2,\ m)) = \text{ndiv}(v_2,\ \text{expt}(2,\ m))
\end{array}
$$

Repeatedly Skolemizing and flattening,
Using lemma bits_equal3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -5 with the terms: (n!1 + m!1),
Simplifying, rewriting, and recording with decision procedures,
Rewriting using bit_set_ndiv_expt2, matching in *,
Rewriting using bit_set_ndiv_expt2, matching in *,
This completes the proof of `bits_equal4`.
Q.E.D.

C.16.9 Bitops.clear_bit_TCC1

Terse proof for `clear_bit_TCC1`.
`clear_bit_TCC1`:

$$
\begin{array}{ll}
\{1\} & \forall\ (n\colon\ \text{nat},\ \text{val}\colon\ \text{nat})\colon\ \text{bit\_set?}(n,\ \text{val}) \supset \text{val} - \text{expt}(2,\ n) \geq 0
\end{array}
$$

Repeatedly Skolemizing and flattening,
Expanding the definition of bit_set?,
Expanding the definition of cut_bit,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Adding type constraints for ndiv(val!1, expt(2, n!1)),
Repeatedly Skolemizing and flattening,
Replacing using formula -4,
Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `clear_bit_TCC1`.
Q.E.D.

C.16.10 Bitops.change_bit_single_bit_op

Terse proof for `change_bit_single_bit_op`.
`change_bit_single_bit_op`:

```
{1}  ∀ (n, m, v: nat, c: ⌊nat → bool⌋):
       n ≠ m ⊃ bit_set?(n, v) = bit_set?(n, change_bit(v, c)(m))
```

Installing automatic rewrites from: change_bit set_bit clear_bit bit_set? cut_bit
Repeatedly Skolemizing and flattening,
Expanding the definition of change_bit,
Expanding the definition of clear_bit,
Expanding the definition of set_bit,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on rem(2)(ndiv(v!1, expt(2, m!1))) = 1,
we get 2 subgoals:
`change_bit_single_bit_op.1`:

```
{-1}  rem(2)(ndiv(v', expt(2, m'))) = 1
{-2}  n' ≥ 0
{-3}  m' ≥ 0
{-4}  v' ≥ 0
{1}   n' = m'
{2}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
       (rem(2)
            (ndiv(IF c'(m')
                     THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1
                             THEN v'
                             ELSE expt(2, m') + v'
                             ENDIF
                     ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1
                             THEN v' − expt(2, m')
                             ELSE v'
                             ENDIF
                     ENDIF,
                  expt(2, n')))
         = 1)
```

Case splitting on c!1(m!1),
we get 2 subgoals:
`change_bit_single_bit_op.1.1:`

```
{-1}  c'(m')
{-2}  rem(2)(ndiv(v', expt(2, m'))) = 1
{-3}  n' ≥ 0
{-4}  m' ≥ 0
{-5}  v' ≥ 0
------------------------------------------------------------------------
{1}   n' = m'
{2}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
       (rem(2)
            (ndiv(IF c'(m')
                      THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1
                              THEN v'
                             ELSE expt(2, m') + v'
                             ENDIF
                     ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1
                             THEN v' - expt(2, m')
                            ELSE v'
                            ENDIF
                     ENDIF,
                     expt(2, n')))
          = 1)
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.1.1`.
`change_bit_single_bit_op.1.2:`

```
{-1}  rem(2)(ndiv(v', expt(2, m'))) = 1
{-2}  n' ≥ 0
{-3}  m' ≥ 0
{-4}  v' ≥ 0
------------------------------------------------------------------------
{1}   c'(m')
{2}   n' = m'
{3}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
       (rem(2)
            (ndiv(IF c'(m')
                      THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1
                              THEN v'
                             ELSE expt(2, m') + v'
                             ENDIF
                     ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1
                             THEN v' - expt(2, m')
                            ELSE v'
                            ENDIF
                     ENDIF,
                     expt(2, n')))
          = 1)
```

Simplifying, rewriting, and recording with decision procedures,
Case splitting on v!1 - expt(2, m!1) >= 0,
we get 2 subgoals:

`change_bit_single_bit_op.1.2.1:`

{-1} $v' - \text{expt}(2,\ m') \geq 0$
{-2} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 1$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $v' \geq 0$

{1} $c'(m')$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 =$
$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n'))) = 1)$

Case splitting on n!1 > m!1,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.1:`

{-1} $n' > m'$
{-2} $v' - \text{expt}(2,\ m') \geq 0$
{-3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 1$
{-4} $n' \geq 0$
{-5} $m' \geq 0$
{-6} $v' \geq 0$

{1} $c'(m')$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 =$
$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n'))) = 1)$

Rewriting using rem_def, matching in *,

Repeatedly Skolemizing and flattening,

Using lemma ndiv_times_2,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.1.1:`

{-1} $\text{ndiv}(v',\ \text{expt}(2,\ m') \times 2 \times \text{expt}(2,\ n' - m' - 1)) =$
$\text{ndiv}(\text{ndiv}(v',\ \text{expt}(2,\ m')),\ 2 \times \text{expt}(2,\ n' - m' - 1))$
{-2} $\text{integer\_pred}(q')$
{-3} $n' > m'$
{-4} $v' - \text{expt}(2,\ m') \geq 0$
{-5} $\text{ndiv}(v',\ \text{expt}(2,\ m')) = 1 + 2 \times q'$
{-6} $n' \geq 0$
{-7} $m' \geq 0$
{-8} $v' \geq 0$

{1} $c'(m')$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 =$
$(\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n'))) = 1)$

Using lemma ndiv_times_2,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.1.1.1:`

{-1} $\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-2} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3} $\mathrm{integer\_pred}(q')$
{-4} $n' > m'$
{-5} $v' - \mathrm{expt}(2,\ m') \geq 0$
{-6} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')) = 1 + 2 \times q'$
{-7} $n' \geq 0$
{-8} $m' \geq 0$
{-9} $v' \geq 0$

{1} $c'(m')$
{2} $n' = m'$
{3} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 = (\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Case splitting on expt(2, m!1) * (2 * expt(2, n!1 - m!1 - 1)) = expt(2, n!1),
we get 3 subgoals:
`change_bit_single_bit_op.1.2.1.1.1.1.1:`

{-1} $\mathrm{expt}(2,\ m') \times (2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{expt}(2,\ n')$
{-2} $\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-4} $\mathrm{integer\_pred}(q')$
{-5} $n' > m'$
{-6} $v' - \mathrm{expt}(2,\ m') \geq 0$
{-7} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')) = 1 + 2 \times q'$
{-8} $n' \geq 0$
{-9} $m' \geq 0$
{-10} $v' \geq 0$

{1} $c'(m')$
{2} $n' = m'$
{3} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 = (\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Replacing using formula -1,
Replacing using formula -2,
Replacing using formula -3,
Using lemma ndiv_minus_2,
Rewriting using divides_reflexive, matching in *,
Rewriting using ndiv_self, matching in *,
Replacing using formula -1,
Replacing using formula -8,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using ndiv_times_2, matching in * where b gets 2, c gets expt(2, -1 - m!1 + n!1),
Rewriting using ndiv_times_2, matching in * where b gets 2, c gets expt(2, -1 - m!1 + n!1),
Using lemma ndiv_plus_mod,

Splitting conjunctions,
we get 2 subgoals:
`change_bit_single_bit_op.1.2.1.1.1.1.1.1:`

{-1} $\text{ndiv}(2 \times q' + 1,\ 2) = \text{ndiv}(2 \times q',\ 2)$
{-2} $\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ m')) = 2 \times q'$
{-3} $2 \times \text{expt}(2,\ -1 - m' + n') \times \text{expt}(2,\ m') = \text{expt}(2,\ n')$
{-4} $\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n')) =$
$\text{ndiv}(\text{ndiv}(2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))$
{-5} $\text{ndiv}(v',\ \text{expt}(2,\ n')) =$
$\text{ndiv}(\text{ndiv}(1 + 2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))$
{-6} $\text{integer\_pred}(q')$
{-7} $n' > m'$
{-8} $v' - \text{expt}(2,\ m') \geq 0$
{-9} $\text{ndiv}(v',\ \text{expt}(2,\ m')) = 1 + 2 \times q'$
{-10} $n' \geq 0$
{-11} $m' \geq 0$
{-12} $v' \geq 0$

{1} $c'(m')$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(\text{ndiv}(1 + 2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))) =$
1
$=$
$(\text{rem}(2)(\text{ndiv}(\text{ndiv}(2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))) = 1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.1.1.1`.
`change_bit_single_bit_op.1.2.1.1.1.1.1.2:`

{-1} $\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ m')) = 2 \times q'$
{-2} $2 \times \text{expt}(2,\ -1 - m' + n') \times \text{expt}(2,\ m') = \text{expt}(2,\ n')$
{-3} $\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n')) =$
$\text{ndiv}(\text{ndiv}(2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))$
{-4} $\text{ndiv}(v',\ \text{expt}(2,\ n')) =$
$\text{ndiv}(\text{ndiv}(1 + 2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))$
{-5} $\text{integer\_pred}(q')$
{-6} $n' > m'$
{-7} $v' - \text{expt}(2,\ m') \geq 0$
{-8} $\text{ndiv}(v',\ \text{expt}(2,\ m')) = 1 + 2 \times q'$
{-9} $n' \geq 0$
{-10} $m' \geq 0$
{-11} $v' \geq 0$

{1} $\text{divides}(2,\ 2 \times q')$
{2} $c'(m')$
{3} $n' = m'$
{4} $\text{rem}(2)(\text{ndiv}(\text{ndiv}(1 + 2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))) =$
1
$=$
$(\text{rem}(2)(\text{ndiv}(\text{ndiv}(2 \times q',\ 2),\ \text{expt}(2,\ -1 - m' + n'))) = 1)$

Expanding the definition of divides,
Instantiating quantified variables,
This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.1.1.2`.

`change_bit_single_bit_op.1.2.1.1.1.1.2:`

{-1} $\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-2} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3} $\mathrm{integer\_pred}(q')$
{-4} $n' > m'$
{-5} $v' - \mathrm{expt}(2,\ m') \geq 0$
{-6} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')) = 1 + 2 \times q'$
{-7} $n' \geq 0$
{-8} $m' \geq 0$
{-9} $v' \geq 0$

{1} $\mathrm{expt}(2,\ m') \times (2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{expt}(2,\ n')$
{2} $c'(m')$
{3} $n' = m'$
{4} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 = (\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Keeping (-4 1) and hiding *,

Expanding the definition of expt,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expt_plus_aux,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.1.2`.

`change_bit_single_bit_op.1.2.1.1.1.1.3:`

{-1} $\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-2} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3} $\mathrm{integer\_pred}(q')$
{-4} $n' > m'$
{-5} $v' - \mathrm{expt}(2,\ m') \geq 0$
{-6} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')) = 1 + 2 \times q'$
{-7} $n' \geq 0$
{-8} $m' \geq 0$
{-9} $v' \geq 0$

{1} $-1 - m' + n' \geq 0$
{2} $c'(m')$
{3} $n' = m'$
{4} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 = (\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.1.3`.

`change_bit_single_bit_op.1.2.1.1.1.2:`

{-1}	$\mathrm{ndiv}(v', \mathrm{expt}(2, m') \times 2 \times \mathrm{expt}(2, n' - m' - 1)) =$ $\mathrm{ndiv}(\mathrm{ndiv}(v', \mathrm{expt}(2, m')), 2 \times \mathrm{expt}(2, n' - m' - 1))$
{-2}	$\mathrm{integer\_pred}(q')$
{-3}	$n' > m'$
{-4}	$v' - \mathrm{expt}(2, m') \geq 0$
{-5}	$\mathrm{ndiv}(v', \mathrm{expt}(2, m')) = 1 + 2 \times q'$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$-1 - m' + n' \geq 0$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2, m'), \mathrm{expt}(2, n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.1.1.2`.

`change_bit_single_bit_op.1.2.1.1.2:`

{-1}	$\mathrm{integer\_pred}(q')$
{-2}	$n' > m'$
{-3}	$v' - \mathrm{expt}(2, m') \geq 0$
{-4}	$\mathrm{ndiv}(v', \mathrm{expt}(2, m')) = 1 + 2 \times q'$
{-5}	$n' \geq 0$
{-6}	$m' \geq 0$
{-7}	$v' \geq 0$
{1}	$-1 - m' + n' \geq 0$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2, m'), \mathrm{expt}(2, n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.1.2`.

`change_bit_single_bit_op.1.2.1.2:`

{-1}	$v' - \mathrm{expt}(2, m') \geq 0$
{-2}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$n' > m'$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2, m'), \mathrm{expt}(2, n'))) = 1)$

Using lemma ndiv_minus_2,

Splitting conjunctions,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.2.1:`

{-1}	$\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n')) =$ $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n')) - \mathrm{ndiv}(\mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))$
{-2}	$v' - \mathrm{expt}(2,\ m') \geq 0$
{-3}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m'))) = 1$
{-4}	$n' \geq 0$
{-5}	$m' \geq 0$
{-6}	$v' \geq 0$
{1}	$n' > m'$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Replacing using formula -1,
Rewriting using ndiv_expt_expt, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Using lemma rem_diff2,
Using lemma rem_multiple1,
Expanding the definition of expt,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.1.2.1.2.1`.
`change_bit_single_bit_op.1.2.1.2.2:`

{-1}	$v' - \mathrm{expt}(2,\ m') \geq 0$
{-2}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\mathrm{divides}(\mathrm{expt}(2,\ n'),\ \mathrm{expt}(2,\ m'))$
{2}	$n' > m'$
{3}	$c'(m')$
{4}	$n' = m'$
{5}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Expanding the definition of divides,
Instantiating the top quantifier in 1 with the terms: (expt(2, m!1 - n!1)),
we get 2 subgoals:
`change_bit_single_bit_op.1.2.1.2.2.1:`

{-1}	$v' - \mathrm{expt}(2,\ m') \geq 0$
{-2}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m'))) = 1$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\mathrm{expt}(2,\ m') = \mathrm{expt}(2,\ n') \times \mathrm{expt}(2,\ m' - n')$
{2}	$n' > m'$
{3}	$c'(m')$
{4}	$n' = m'$
{5}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ n'))) = 1)$

Using lemma expt_plus_aux,

we get 2 subgoals:

`change_bit_single_bit_op.1.2.1.2.2.1.1:`

$$
\begin{array}{ll}
\{\text{-1}\} & \text{expt}(2,\ n' + m' - n') = \text{expt}(2,\ n') \times \text{expt}(2,\ m' - n') \\
\{\text{-2}\} & v' - \text{expt}(2,\ m') \geq 0 \\
\{\text{-3}\} & \text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 1 \\
\{\text{-4}\} & n' \geq 0 \\
\{\text{-5}\} & m' \geq 0 \\
\{\text{-6}\} & v' \geq 0 \\
\hline
\{1\} & \text{expt}(2,\ m') = \text{expt}(2,\ n') \times \text{expt}(2,\ m' - n') \\
\{2\} & n' > m' \\
\{3\} & c'(m') \\
\{4\} & n' = m' \\
\{5\} & \text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 = \\
 & \quad (\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n'))) = 1)
\end{array}
$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.2.1.1`.

`change_bit_single_bit_op.1.2.1.2.2.1.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & v' - \text{expt}(2,\ m') \geq 0 \\
\{\text{-2}\} & \text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 1 \\
\{\text{-3}\} & n' \geq 0 \\
\{\text{-4}\} & m' \geq 0 \\
\{\text{-5}\} & v' \geq 0 \\
\hline
\{1\} & m' - n' \geq 0 \\
\{2\} & \text{expt}(2,\ m') = \text{expt}(2,\ n') \times \text{expt}(2,\ m' - n') \\
\{3\} & n' > m' \\
\{4\} & c'(m') \\
\{5\} & n' = m' \\
\{6\} & \text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 = \\
 & \quad (\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n'))) = 1)
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.2.1.2`.

`change_bit_single_bit_op.1.2.1.2.2.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & v' - \text{expt}(2,\ m') \geq 0 \\
\{\text{-2}\} & \text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 1 \\
\{\text{-3}\} & n' \geq 0 \\
\{\text{-4}\} & m' \geq 0 \\
\{\text{-5}\} & v' \geq 0 \\
\hline
\{1\} & m' - n' \geq 0 \\
\{2\} & n' > m' \\
\{3\} & c'(m') \\
\{4\} & n' = m' \\
\{5\} & \text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 = \\
 & \quad (\text{rem}(2)(\text{ndiv}(v' - \text{expt}(2,\ m'),\ \text{expt}(2,\ n'))) = 1)
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `change_bit_single_bit_op.1.2.1.2.2.2`.

`change_bit_single_bit_op.1.2.2:`

{-1}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$v' \geq 0$
{1}	$v' - \mathrm{expt}(2, m') \geq 0$
{2}	$c'(m')$
{3}	$n' = m'$
{4}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(v' - \mathrm{expt}(2, m'), \mathrm{expt}(2, n'))) = 1)$

Hiding formulas: (-2 2 3 4),

Case splitting on v!1 < expt(2, m!1),

we get 2 subgoals:

`change_bit_single_bit_op.1.2.2.1:`

{-1}	$v' < \mathrm{expt}(2, m')$
{-2}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1$
{-3}	$m' \geq 0$
{-4}	$v' \geq 0$
{1}	$v' - \mathrm{expt}(2, m') \geq 0$

Rewriting using ndiv_bigger, matching in *,

Rewriting using rem_zero, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.2.1`.

`change_bit_single_bit_op.1.2.2.2:`

{-1}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1$
{-2}	$m' \geq 0$
{-3}	$v' \geq 0$
{1}	$v' < \mathrm{expt}(2, m')$
{2}	$v' - \mathrm{expt}(2, m') \geq 0$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.1.2.2.2`.

`change_bit_single_bit_op.2:`

```
{-1}  n' >= 0
{-2}  m' >= 0
{-3}  v' >= 0
|-------
{1}   rem(2)(ndiv(v', expt(2, m'))) = 1
{2}   n' = m'
{3}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
       (rem(2)
            (ndiv(IF c'(m')
                     THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1
                             THEN v'
                          ELSE expt(2, m') + v'
                          ENDIF
                  ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1
                          THEN v' - expt(2, m')
                       ELSE v'
                       ENDIF
                  ENDIF,
                  expt(2, n')))
          = 1)
```

Case splitting on c!1(m!1),

we get 2 subgoals:

`change_bit_single_bit_op.2.1:`

```
{-1}  c'(m')
{-2}  n' >= 0
{-3}  m' >= 0
{-4}  v' >= 0
|-------
{1}   rem(2)(ndiv(v', expt(2, m'))) = 1
{2}   n' = m'
{3}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
       (rem(2)
            (ndiv(IF c'(m')
                     THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1
                             THEN v'
                          ELSE expt(2, m') + v'
                          ENDIF
                  ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1
                          THEN v' - expt(2, m')
                       ELSE v'
                       ENDIF
                  ENDIF,
                  expt(2, n')))
          = 1)
```

Simplifying, rewriting, and recording with decision procedures,

Case splitting on n!1 > m!1,

we get 2 subgoals:

`change_bit_single_bit_op.2.1.1:`

{-1}	$n' > m'$
{-2}	$c'(m')$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$v' \geq 0$
{1}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1$
{2}	$n' = m'$
{3}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2, m') + v', \mathrm{expt}(2, n'))) = 1)$

Case splitting on rem(2)(ndiv(v!1, expt(2, m!1))) = 0,

we get 2 subgoals:

`change_bit_single_bit_op.2.1.1.1:`

{-1}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 0$
{-2}	$n' > m'$
{-3}	$c'(m')$
{-4}	$n' \geq 0$
{-5}	$m' \geq 0$
{-6}	$v' \geq 0$
{1}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1$
{2}	$n' = m'$
{3}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2, m') + v', \mathrm{expt}(2, n'))) = 1)$

Hiding formulas: 1,

Using lemma ndiv_times_2,

we get 2 subgoals:

`change_bit_single_bit_op.2.1.1.1.1:`

{-1}	$\mathrm{ndiv}(v', \mathrm{expt}(2, m') \times 2 \times \mathrm{expt}(2, n' - m' - 1)) =$ $\mathrm{ndiv}(\mathrm{ndiv}(v', \mathrm{expt}(2, m')), 2 \times \mathrm{expt}(2, n' - m' - 1))$
{-2}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 0$
{-3}	$n' > m'$
{-4}	$c'(m')$
{-5}	$n' \geq 0$
{-6}	$m' \geq 0$
{-7}	$v' \geq 0$
{1}	$n' = m'$
{2}	$\mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2, m') + v', \mathrm{expt}(2, n'))) = 1)$

Using lemma ndiv_times_2,

we get 2 subgoals:

`change_bit_single_bit_op.2.1.1.1.1.1:`

{-1} $\mathrm{ndiv}(v' + \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
 =
 $\mathrm{ndiv}(\mathrm{ndiv}(v' + \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),$
 $2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-2} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) =$
 $\mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m'))) = 0$
{-4} $n' > m'$
{-5} $c'(m')$
{-6} $n' \geq 0$
{-7} $m' \geq 0$
{-8} $v' \geq 0$

{1} $n' = m'$
{2} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 =$
 $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2,\ m') + v',\ \mathrm{expt}(2,\ n'))) = 1)$

Case splitting on expt(2, m!1) * (2 * expt(2, n!1 - m!1 - 1)) = expt(2, n!1),
we get 3 subgoals:
`change_bit_single_bit_op.2.1.1.1.1.1.1:`

{-1} $\mathrm{expt}(2,\ m') \times (2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{expt}(2,\ n')$
{-2} $\mathrm{ndiv}(v' + \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
 =
 $\mathrm{ndiv}(\mathrm{ndiv}(v' + \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),$
 $2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3} $\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) =$
 $\mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-4} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m'))) = 0$
{-5} $n' > m'$
{-6} $c'(m')$
{-7} $n' \geq 0$
{-8} $m' \geq 0$
{-9} $v' \geq 0$

{1} $n' = m'$
{2} $\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 =$
 $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2,\ m') + v',\ \mathrm{expt}(2,\ n'))) = 1)$

Replacing using formula -1,
Replacing using formula -2,
Replacing using formula -3,
Using lemma ndiv_plus_2,
Rewriting using divides_reflexive, matching in *,
Rewriting using ndiv_self, matching in *,
Replacing using formula -1,
Hiding formulas: (-1 -2 -3 -4),
Rewriting using rem_def, matching in *,
Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Rewriting using ndiv_times_2, matching in * where b gets 2, c gets expt(2, -1 - m!1 + n!1),
Rewriting using ndiv_times_2, matching in * where b gets 2, c gets expt(2, -1 - m!1 + n!1),
Using lemma ndiv_plus_mod,
Splitting conjunctions,
we get 2 subgoals:

`change_bit_single_bit_op.2.1.1.1.1.1.1.1:`

{-1}	$\mathrm{ndiv}(2 \times q' + 1,\ 2) = \mathrm{ndiv}(2 \times q',\ 2)$
{-2}	$\mathrm{integer\_pred}(q')$
{-3}	$\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')) = 2 \times q'$
{-4}	$n' > m'$
{-5}	$c'(m')$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$n' = m'$
{2}	$\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{ndiv}(2 \times q',\ 2),\ \mathrm{expt}(2,\ -1 - m' + n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{ndiv}(1 + 2 \times q',\ 2),\ \mathrm{expt}(2,\ -1 - m' + n'))) = 1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.1.1.1`.
`change_bit_single_bit_op.2.1.1.1.1.1.1.2:`

{-1}	$\mathrm{integer\_pred}(q')$
{-2}	$\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')) = 2 \times q'$
{-3}	$n' > m'$
{-4}	$c'(m')$
{-5}	$n' \geq 0$
{-6}	$m' \geq 0$
{-7}	$v' \geq 0$
{1}	$\mathrm{divides}(2,\ 2 \times q')$
{2}	$n' = m'$
{3}	$\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{ndiv}(2 \times q',\ 2),\ \mathrm{expt}(2,\ -1 - m' + n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{ndiv}(1 + 2 \times q',\ 2),\ \mathrm{expt}(2,\ -1 - m' + n'))) = 1)$

Expanding the definition of divides,
Instantiating quantified variables,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.1.1.1.2:`

{-1}	$\mathrm{ndiv}(v' + \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1))$ $=$ $\mathrm{ndiv}(\mathrm{ndiv}(v' + \mathrm{expt}(2,\ m'),\ \mathrm{expt}(2,\ m')),$ $2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-2}	$\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m') \times 2 \times \mathrm{expt}(2,\ n' - m' - 1)) =$ $\mathrm{ndiv}(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m')),\ 2 \times \mathrm{expt}(2,\ n' - m' - 1))$
{-3}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ m'))) = 0$
{-4}	$n' > m'$
{-5}	$c'(m')$
{-6}	$n' \geq 0$
{-7}	$m' \geq 0$
{-8}	$v' \geq 0$
{1}	$\mathrm{expt}(2,\ m') \times (2 \times \mathrm{expt}(2,\ n' - m' - 1)) = \mathrm{expt}(2,\ n')$
{2}	$n' = m'$
{3}	$\mathrm{rem}(2)(\mathrm{ndiv}(v',\ \mathrm{expt}(2,\ n'))) = 1 =$ $(\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2,\ m') + v',\ \mathrm{expt}(2,\ n'))) = 1)$

Keeping (-4 1) and hiding *,

Expanding the definition of expt,
Simplifying, rewriting, and recording with decision procedures,
Using lemma expt_plus_aux,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.1.1.1.3`:

{-1} $\text{ndiv}(v' + \text{expt}(2,\ m'),\ \text{expt}(2,\ m') \times 2 \times \text{expt}(2,\ n' - m' - 1)) = \text{ndiv}(\text{ndiv}(v' + \text{expt}(2,\ m'),\ \text{expt}(2,\ m')),\ 2 \times \text{expt}(2,\ n' - m' - 1))$
{-2} $\text{ndiv}(v',\ \text{expt}(2,\ m') \times 2 \times \text{expt}(2,\ n' - m' - 1)) = \text{ndiv}(\text{ndiv}(v',\ \text{expt}(2,\ m')),\ 2 \times \text{expt}(2,\ n' - m' - 1))$
{-3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 0$
{-4} $n' > m'$
{-5} $c'(m')$
{-6} $n' \geq 0$
{-7} $m' \geq 0$
{-8} $v' \geq 0$

{1} $-1 - m' + n' \geq 0$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 = (\text{rem}(2)(\text{ndiv}(\text{expt}(2,\ m') + v',\ \text{expt}(2,\ n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.1.3`.
`change_bit_single_bit_op.2.1.1.1.1.2`:

{-1} $\text{ndiv}(v',\ \text{expt}(2,\ m') \times 2 \times \text{expt}(2,\ n' - m' - 1)) = \text{ndiv}(\text{ndiv}(v',\ \text{expt}(2,\ m')),\ 2 \times \text{expt}(2,\ n' - m' - 1))$
{-2} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 0$
{-3} $n' > m'$
{-4} $c'(m')$
{-5} $n' \geq 0$
{-6} $m' \geq 0$
{-7} $v' \geq 0$

{1} $-1 - m' + n' \geq 0$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 = (\text{rem}(2)(\text{ndiv}(\text{expt}(2,\ m') + v',\ \text{expt}(2,\ n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.1.2`:

{-1} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ m'))) = 0$
{-2} $n' > m'$
{-3} $c'(m')$
{-4} $n' \geq 0$
{-5} $m' \geq 0$
{-6} $v' \geq 0$

{1} $-1 - m' + n' \geq 0$
{2} $n' = m'$
{3} $\text{rem}(2)(\text{ndiv}(v',\ \text{expt}(2,\ n'))) = 1 = (\text{rem}(2)(\text{ndiv}(\text{expt}(2,\ m') + v',\ \text{expt}(2,\ n'))) = 1)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.1.1.2`.
`change_bit_single_bit_op.2.1.1.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & n' > m' \\
\{\text{-}2\} & c'(m') \\
\{\text{-}3\} & n' \geq 0 \\
\{\text{-}4\} & m' \geq 0 \\
\{\text{-}5\} & v' \geq 0 \\
\hline
\{1\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 0 \\
\{2\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1 \\
\{3\} & n' = m' \\
\{4\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 = \\
 & \quad (\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2, m') + v', \mathrm{expt}(2, n'))) = 1)
\end{array}
$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_single_bit_op.2.1.1.2`.
`change_bit_single_bit_op.2.1.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & c'(m') \\
\{\text{-}2\} & n' \geq 0 \\
\{\text{-}3\} & m' \geq 0 \\
\{\text{-}4\} & v' \geq 0 \\
\hline
\{1\} & n' > m' \\
\{2\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1 \\
\{3\} & n' = m' \\
\{4\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 = \\
 & \quad (\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2, m') + v', \mathrm{expt}(2, n'))) = 1)
\end{array}
$$

Using lemma ndiv_plus_2,
Splitting conjunctions,
we get 2 subgoals:
`change_bit_single_bit_op.2.1.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \mathrm{ndiv}(v' + \mathrm{expt}(2, m'), \mathrm{expt}(2, n')) = \\
 & \quad \mathrm{ndiv}(v', \mathrm{expt}(2, n')) + \mathrm{ndiv}(\mathrm{expt}(2, m'), \mathrm{expt}(2, n')) \\
\{\text{-}2\} & c'(m') \\
\{\text{-}3\} & n' \geq 0 \\
\{\text{-}4\} & m' \geq 0 \\
\{\text{-}5\} & v' \geq 0 \\
\hline
\{1\} & n' > m' \\
\{2\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, m'))) = 1 \\
\{3\} & n' = m' \\
\{4\} & \mathrm{rem}(2)(\mathrm{ndiv}(v', \mathrm{expt}(2, n'))) = 1 = \\
 & \quad (\mathrm{rem}(2)(\mathrm{ndiv}(\mathrm{expt}(2, m') + v', \mathrm{expt}(2, n'))) = 1)
\end{array}
$$

Rewriting using ndiv_expt_expt, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Using lemma rem_sum1,
Using lemma rem_multiple1,
Expanding the definition of expt,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.2.1`.

`change_bit_single_bit_op.2.1.2.2:`

{-1} $c'(m')$
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} $v' \geq 0$

{1} divides(expt(2, n'), expt(2, m'))
{2} $n' > m'$
{3} rem(2)(ndiv(v', expt(2, m'))) = 1
{4} $n' = m'$
{5} rem(2)(ndiv(v', expt(2, n'))) = 1 =
(rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)

Expanding the definition of divides,

Instantiating the top quantifier in 1 with the terms: (expt(2, m!1 - n!1)),

we get 2 subgoals:

`change_bit_single_bit_op.2.1.2.2.1:`

{-1} $c'(m')$
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} $v' \geq 0$

{1} expt(2, m') = expt(2, n') $\times$ expt(2, $m' - n'$)
{2} $n' > m'$
{3} rem(2)(ndiv(v', expt(2, m'))) = 1
{4} $n' = m'$
{5} rem(2)(ndiv(v', expt(2, n'))) = 1 =
(rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)

Using lemma expt_plus_aux,

we get 2 subgoals:

`change_bit_single_bit_op.2.1.2.2.1.1:`

{-1} expt(2, $n' + m' - n'$) = expt(2, n') $\times$ expt(2, $m' - n'$)
{-2} $c'(m')$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $v' \geq 0$

{1} expt(2, m') = expt(2, n') $\times$ expt(2, $m' - n'$)
{2} $n' > m'$
{3} rem(2)(ndiv(v', expt(2, m'))) = 1
{4} $n' = m'$
{5} rem(2)(ndiv(v', expt(2, n'))) = 1 =
(rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.2.1.2.2.1.1`.

`change_bit_single_bit_op.2.1.2.2.1.2:`

```
{-1}  c'(m')
{-2}  n' ≥ 0
{-3}  m' ≥ 0
{-4}  v' ≥ 0
------------------------------------------------------------
{1}   m' - n' ≥ 0
{2}   expt(2, m') = expt(2, n') × expt(2, m' - n')
{3}   n' > m'
{4}   rem(2)(ndiv(v', expt(2, m'))) = 1
{5}   n' = m'
{6}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
        (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.2.2.1.2`.
`change_bit_single_bit_op.2.1.2.2.2:`

```
{-1}  c'(m')
{-2}  n' ≥ 0
{-3}  m' ≥ 0
{-4}  v' ≥ 0
------------------------------------------------------------
{1}   m' - n' ≥ 0
{2}   n' > m'
{3}   rem(2)(ndiv(v', expt(2, m'))) = 1
{4}   n' = m'
{5}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
        (rem(2)(ndiv(expt(2, m') + v', expt(2, n'))) = 1)
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bit_single_bit_op.2.1.2.2.2`.
`change_bit_single_bit_op.2.2:`

```
{-1}  n' ≥ 0
{-3}  m' ≥ 0
{-4}  v' ≥ 0
------------------------------------------------------------
{1}   c'(m')
{2}   rem(2)(ndiv(v', expt(2, m'))) = 1
{3}   n' = m'
{4}   rem(2)(ndiv(v', expt(2, n'))) = 1 =
        (rem(2)
              (ndiv(IF c'(m')
                       THEN IF rem(2)(ndiv(v', expt(2, m'))) = 1
                               THEN v'
                               ELSE expt(2, m') + v'
                               ENDIF
                       ELSE IF rem(2)(ndiv(v', expt(2, m'))) = 1
                               THEN v' - expt(2, m')
                               ELSE v'
                               ENDIF
                       ENDIF,
                       expt(2, n')))
          = 1)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `change_bit_single_bit_op.2.2`.
Q.E.D.

C.16.11 Bitops.change_bit_result

Terse proof for `change_bit_result`.
`change_bit_result`:

{1} $\forall$ (n, v: nat, c: $\lfloor$nat $\to$ bool$\rfloor$): bit_set?(n, change_bit(v, c)(n)) = $c(n)$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:
`change_bit_result.1`:

{-1} $n' \geq 0$
{-2} $v' \geq 0$
{-3} rem(2)(ndiv(v', expt(2, n'))) = 1
{-4} rem(2)(ndiv($v' - $expt(2, n'), expt(2, n'))) = 1

{1} $c'(n')$

Using lemma ndiv_minus_2,
Rewriting using divides_reflexive, matching in *,
Rewriting using ndiv_self, matching in *,
Using lemma rem_add_one,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_result.1`.
`change_bit_result.2`:

{-1} $n' \geq 0$
{-2} $v' \geq 0$
{-3} $c'(n')$

{1} rem(2)(ndiv(v', expt(2, n'))) = 1
{2} rem(2)(ndiv(expt(2, n') + v', expt(2, n'))) = 1

Using lemma ndiv_plus_1,
Rewriting using divides_reflexive, matching in *,
Rewriting using ndiv_self, matching in *,
Using lemma rem_add_one,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bit_result.2`.
Q.E.D.

C.16.12 Bitops.change_bits_TCC1

Terse proof for `change_bits_TCC1`.
`change_bits_TCC1`:

{1} $\forall$ (n: nat): $\neg$ $n = 0 \supset n - 1 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `change_bits_TCC1`.
Q.E.D.

C.16.13 Bitops.change_bits_TCC2

Terse proof for `change_bits_TCC2`.

`change_bits_TCC2:`

```
{1}  ∀ (n: nat): ¬ n = 0 ⊃ n − 1 < n
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `change_bits_TCC2`.
Q.E.D.

C.16.14 Bitops.change_bits_result

Terse proof for `change_bits_result`.

`change_bits_result:`

```
{1}  ∀ (m, n: nat, val: nat, c: [nat → bool]):
       m ≤ n ⊃ bit_set?(m, change_bits(val, c)(n)) = c(m)
```

Inducting on n on formula 1,
we get 2 subgoals:

`change_bits_result.1:`

```
{1}  ∀ (m, val: nat, c: [nat → bool]):
       m ≤ 0 ⊃ bit_set?(m, change_bits(val, c)(0)) = c(m)
```

Repeatedly Skolemizing and flattening,
Case splitting on m!1 = 0,
we get 2 subgoals:

`change_bits_result.1.1:`

```
{-1}  m′ = 0
{-2}  m′ ≥ 0
{-3}  val′ ≥ 0
{-4}  m′ ≤ 0
-------------------------------------------------
 {1}  bit_set?(m′, change_bits(val′, c′)(0)) = c′(m′)
```

Replacing using formula -1,
Expanding the definition of change_bits,
Using lemma change_bit_result,
This completes the proof of `change_bits_result.1.1`.

`change_bits_result.1.2:`

```
{-1}  m′ ≥ 0
{-2}  val′ ≥ 0
{-3}  m′ ≤ 0
-------------------------------------------------
 {1}  m′ = 0
 {2}  bit_set?(m′, change_bits(val′, c′)(0)) = c′(m′)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_result.1.2`.

`change_bits_result.2:`

{1} ∀ j:
(∀ (m, val: nat, c: [nat → bool]):
$m \leq j \supset$ bit_set?(m, change_bits(val, c)(j)) = $c(m)$)
⊃
(∀ (m, val: nat, c: [nat → bool]):
$m \leq j+1 \supset$ bit_set?(m, change_bits(val, c)($j+1$)) = $c(m)$)

Repeatedly Skolemizing and flattening,
Expanding the definition of change_bits,
Case splitting on m!1 = j!1 + 1,
we get 2 subgoals:
`change_bits_result.2.1:`

{-1} $m' = j' + 1$
{-2} $m' \geq 0$
{-3} val$' \geq 0$
{-4} $j' \geq 0$
{-5} ∀ (m, val: nat, c: [nat → bool]):
$m \leq j' \supset$ bit_set?(m, change_bits(val, c)(j')) = $c(m)$
{-6} $m' \leq j' + 1$

{1} bit_set?(m', change_bit(change_bits(val$'$, c')(j'), c')($1 + j'$)) = $c'(m')$

Using lemma change_bit_result,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bits_result.2.1`.
`change_bits_result.2.2:`

{-1} $m' \geq 0$
{-2} val$' \geq 0$
{-3} $j' \geq 0$
{-4} ∀ (m, val: nat, c: [nat → bool]):
$m \leq j' \supset$ bit_set?(m, change_bits(val, c)(j')) = $c(m)$
{-5} $m' \leq j' + 1$

{1} $m' = j' + 1$
{2} bit_set?(m', change_bit(change_bits(val$'$, c')(j'), c')($1 + j'$)) = $c'(m')$

Using lemma change_bit_single_bit_op,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bits_result.2.2`.
Q.E.D.

C.16.15 Bitops.change_bits_unchanged

Terse proof for `change_bits_unchanged`.
`change_bits_unchanged:`

{1} ∀ (m, n: nat, val: nat, c: [nat → bool]):
$m > n \supset$ bit_set?(m, change_bits(val, c)(n)) = bit_set?(m, val)

Inducting on n on formula 1,
we get 2 subgoals:
`change_bits_unchanged.1`:

{1} ∀ (m, val: nat, c: [nat → bool]):
$m > 0$ ⊃ bit_set?(m, change_bits(val, c)(0)) = bit_set?(m, val)

Repeatedly Skolemizing and flattening,
Expanding the definition of change_bits,
Using lemma change_bit_single_bit_op,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_unchanged.1`.
`change_bits_unchanged.2`:

{1} ∀ j:
(∀ (m, val: nat, c: [nat → bool]):
$m > j$ ⊃ bit_set?(m, change_bits(val, c)(j)) = bit_set?(m, val))
⊃
(∀ (m, val: nat, c: [nat → bool]):
$m > j+1$ ⊃
bit_set?(m, change_bits(val, c)($j+1$)) = bit_set?(m, val))

Repeatedly Skolemizing and flattening,
Expanding the definition of change_bits,
Using lemma change_bit_single_bit_op,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_unchanged.2`.
Q.E.D.

C.16.16 Bitops.change_bits_below_TCC1

Terse proof for `change_bits_below_TCC1`.
`change_bits_below_TCC1`:

{1} ∀ (v: below[expt(2, p)]): $p - 1 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `change_bits_below_TCC1`.
Q.E.D.

C.16.17 Bitops.change_bits_below

Terse proof for `change_bits_below`.
`change_bits_below`:

{1} ∀ (v: below[expt(2, p)], c: [nat → bool]):
change_bits(v, c)($p-1$) < expt(2, p)

Repeatedly Skolemizing and flattening,

Using lemma number_split,
we get 2 subgoals:
`change_bits_below.1:`

{-1} change_bits(v', c')($p-1$) =
  expt(2, p) × ndiv(change_bits(v', c')($p-1$), expt(2, p)) + rem(expt(2, p))(change_bits(v', c')($p-1$))
{-2} $v' <$ expt(2, p)

{1} change_bits(v', c')($p-1$) < expt(2, p)

Replacing using formula -1,
Hiding formulas: -1,
Using lemma bit_split_less,
we get 2 subgoals:
`change_bits_below.1.1:`

{-1} rem(expt(2, p))(change_bits(v', c')($p-1$)) < expt(2, p) ⊃
  (ndiv(change_bits(v', c')($p-1$), expt(2, p)) × expt(2, p) + rem(expt(2, p))(change_bits(v', c')($p-1$))
    < expt(2, $p+0$)
    ≡ ndiv(change_bits(v', c')($p-1$), expt(2, p)) < expt(2, 0))
{-2} $v' <$ expt(2, p)

{1} expt(2, p) × ndiv(change_bits(v', c')($p-1$), expt(2, p)) + rem(expt(2, p))(change_bits(v', c')($p-1$))
  < expt(2, p)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (1 3),
Applying change_bits_unchanged
Instantiating the top quantifier in -1 with the terms: (_ p - 1 v!1 c!1),
Using lemma bits_equal4,
Splitting conjunctions,
we get 2 subgoals:
`change_bits_below.1.1.1:`

{-1} ndiv(change_bits(v', c')($p-1$), expt(2, p)) = ndiv(v', expt(2, p))
{-2} ∀ (m: nat):
    $m > p-1$ ⊃
      bit_set?(m, change_bits(v', c')($p-1$)) = bit_set?(m, v')
{-3} $v' <$ expt(2, p)

{1} ndiv(change_bits(v', c')($p-1$), expt(2, p)) < expt(2, 0)

Replacing using formula -1,
Hiding formulas: (-1 -2),
Rewriting using ndiv_bigger, matching in *,
Expanding the definition of expt,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `change_bits_below.1.1.1`.
`change_bits_below.1.1.2:`

{-1} ∀ (m: nat):
    $m > p-1$ ⊃
      bit_set?(m, change_bits(v', c')($p-1$)) = bit_set?(m, v')
{-2} $v' <$ expt(2, p)

{1} ∀ (n: nat):
    $n \geq p$ ⊃ bit_set?(n, change_bits(v', c')($p-1$)) = bit_set?(n, v')
{2} ndiv(change_bits(v', c')($p-1$), expt(2, p)) < expt(2, 0)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_below.1.1.2`.
`change_bits_below.1.2`:

{-1}	$v' < \text{expt}(2, p)$
{1}	$p - 1 \geq 0$
{2}	$\text{expt}(2, p) \times \text{ndiv}(\text{change\_bits}(v', c')(p-1), \text{expt}(2, p)) + \text{rem}(\text{expt}(2, p))(\text{change\_bits}(v', c')(p$ $< \text{expt}(2, p)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_below.1.2`.
`change_bits_below.2`:

{-1}	$v' < \text{expt}(2, p)$
{1}	$p - 1 \geq 0$
{2}	$\text{change\_bits}(v', c')(p-1) < \text{expt}(2, p)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `change_bits_below.2`.
Q.E.D.

C.16.18 Bitops.binary_not_TCC1

Terse proof for `binary_not_TCC1`.
`binary_not_TCC1`:

{1}	$\forall$ (val: $\text{below}\lfloor\text{expt}(2, p)\rfloor$): $\text{change\_bits}(\text{val}, \lambda\ (n: \text{nat}): \neg\ \text{bit\_set?}(n, \text{val}))(p-1) < \text{expt}(2, p)$

Repeatedly Skolemizing and flattening,
Rewriting using change_bits_below, matching in *,
This completes the proof of `binary_not_TCC1`.
Q.E.D.

C.16.19 Bitops.binary_not_not

Terse proof for `binary_not_not`.
`binary_not_not`:

{1}	$\forall$ (val: $\text{below}\lfloor\text{expt}(2, p)\rfloor$): $\text{binary\_not}(\text{binary\_not}(\text{val})) = \text{val}$

Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of binary_not,
Using lemma change_bits_result,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Using lemma change_bits_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_not_not`.
Q.E.D.

C.16.20 Bitops.binary_and_TCC1

Terse proof for `binary_and_TCC1`.
`binary_and_TCC1`:

{1} $\forall$ (v_1, v_2: below$\lfloor$expt(2, p)$\rfloor$):
change_bits(v_1, λ (n: nat): bit_set?(n, v_1) $\wedge$ bit_set?(n, v_2))($p-1$) $<$
expt(2, p)

Repeatedly Skolemizing and flattening,
Rewriting using change_bits_below, matching in *,
This completes the proof of `binary_and_TCC1`.
Q.E.D.

C.16.21 Bitops.binary_and_not

Terse proof for `binary_and_not`.
`binary_and_not`:

{1} $\forall$ (val: below$\lfloor$expt(2, p)$\rfloor$): binary_and(val, binary_not(val)) $=$ 0

Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of binary_and,
Using lemma change_bits_result,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Expanding the definition of binary_not,
Using lemma change_bits_result,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Expanding the definition of bit_set?,
Using lemma cut_bit_zero,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_and_not`.
Q.E.D.

C.16.22 Bitops.binary_and_full_TCC1

Terse proof for `binary_and_full_TCC1`.
`binary_and_full_TCC1`:

{1} $\forall$ (val: below$\lfloor$expt(2, p)$\rfloor$):
expt(2, p) $-$ 1 $\geq$ 0 $\wedge$ expt(2, p) $-$ 1 $<$ expt(2, p)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_and_full_TCC1`.
Q.E.D.

C.16.23 Bitops.binary_and_full

Terse proof for `binary_and_full`.
`binary_and_full:`

```
{1}  ∀ (val: below⌊expt(2, p)⌋): binary_and(val, expt(2, p) − 1) = val
```

Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
we get 2 subgoals:
`binary_and_full.1:`

```
{-1}  (∀ (n: nat):
         n < p ⊃
           bit_set?(n, binary_and(val′, expt(2, p) − 1)) = bit_set?(n, val′))
       ≡ binary_and(val′, expt(2, p) − 1) = val′
{-2}  val′ < expt(2, p)
{1}   binary_and(val′, expt(2, p) − 1) = val′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Hiding formulas: 2,
Expanding the definition of binary_and,
Using lemma change_bits_result,
we get 2 subgoals:
`binary_and_full.1.1:`

```
{-1}  n′ ≤ p − 1 ⊃
       bit_set?(n′,
                change_bits(val′,
                            λ (n: nat):
                              bit_set?(n, val′) ∧ bit_set?(n, expt(2, p) − 1))
                           (p − 1))
         = (bit_set?(n′, val′) ∧ bit_set?(n′, expt(2, p) − 1))
{-2}  n′ ≥ 0
{-3}  n′ < p
{-4}  val′ < expt(2, p)
{1}   bit_set?(n′,
               change_bits(val′,
                           λ (n: nat):
                             bit_set?(n, val′) ∧ bit_set?(n, expt(2, p) − 1))
                          (p − 1))
       = bit_set?(n′, val′)
```

Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,

Hiding formulas: -1,
Rewriting using bit_set_full_mask, matching in *,
This completes the proof of `binary_and_full.1.1`.
`binary_and_full.1.2:`

```
{-1}  n' >= 0
{-2}  n' < p
{-3}  val' < expt(2, p)
|-------
{1}   FORALL (n: nat): bit_set?(n, val') IMPLIES expt(2, p) - 1 >= 0
{2}   bit_set?(n',
                change_bits(val',
                            LAMBDA (n: nat):
                              bit_set?(n, val') AND bit_set?(n, expt(2, p) - 1))
                           (p - 1))
        = bit_set?(n', val')
```

Repeatedly Skolemizing and flattening,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_and_full.1.2`.
`binary_and_full.2:`

```
{-1}  val' < expt(2, p)
|-------
{1}   expt(2, p) - 1 >= 0
{2}   binary_and(val', expt(2, p) - 1) = val'
```

Expanding the definition of binary_and,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_and_full.2`.
Q.E.D.

C.16.24 Bitops.binary_and_zero_TCC1

Terse proof for `binary_and_zero_TCC1`.
`binary_and_zero_TCC1:`

```
|-------
{1}   FORALL (val: below(ceiling(expt(2, p)))): 0 < expt(2, p)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `binary_and_zero_TCC1`.
Q.E.D.

C.16.25 Bitops.binary_and_zero

Terse proof for `binary_and_zero`.
`binary_and_zero:`

```
|-------
{1}   FORALL (val: below(ceiling(expt(2, p)))): binary_and(val, 0) = 0
```

Repeatedly Skolemizing and flattening,

Using lemma bits_equal2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Hiding formulas: 2,
Expanding the definition of binary_and,
Rewriting using change_bits_result, matching in *,
Rewriting using bit_set_zero_mask, matching in *,
This completes the proof of `binary_and_zero`.
Q.E.D.

C.16.26 Bitops.binary_or_TCC1

Terse proof for `binary_or_TCC1`.
`binary_or_TCC1`:

$$\{1\}\quad \forall\ (v_1,\ v_2\colon \text{below}\lfloor \text{expt}(2,\ p)\rfloor)\colon \text{change\_bits}(v_1,\ \lambda\ (n\colon \text{nat})\colon \text{bit\_set?}(n,\ v_1)\ \vee\ \text{bit\_set?}(n,\ v_2))(p-1)\ <\ \text{expt}(2,\ p)$$

Repeatedly Skolemizing and flattening,
Rewriting using change_bits_below, matching in *,
This completes the proof of `binary_or_TCC1`.
Q.E.D.

C.16.27 Bitops.binary_or_not

Terse proof for `binary_or_not`.
`binary_or_not`:

$$\{1\}\quad \forall\ (\text{val}\colon \text{below}\lfloor \text{expt}(2,\ p)\rfloor)\colon \text{binary\_or}(\text{val},\ \text{binary\_not}(\text{val}))\ =\ \text{expt}(2,\ p)-1$$

Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
we get 2 subgoals:
`binary_or_not.1`:

$$\begin{array}{ll}\{-1\} & (\forall\ (n\colon \text{nat})\colon\ n\ <\ p\ \supset\ \text{bit\_set?}(n,\ \text{binary\_or}(\text{val}',\ \text{binary\_not}(\text{val}')))\ =\ \text{bit\_set?}(n,\ \text{expt}(2,\ p)-1))\\ & \equiv\ \text{binary\_or}(\text{val}',\ \text{binary\_not}(\text{val}'))\ =\ \text{expt}(2,\ p)-1\\ \{-2\} & \text{val}'\ <\ \text{expt}(2,\ p)\\ \hline \{1\} & \text{binary\_or}(\text{val}',\ \text{binary\_not}(\text{val}'))\ =\ \text{expt}(2,\ p)-1\end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of binary_or,
Rewriting using change_bits_result, matching in *,
Expanding the definition of binary_not,
Rewriting using change_bits_result, matching in *,
Rewriting using bit_set_full_mask, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `binary_or_not.1`.
`binary_or_not.2`:

```
{-1}  val′ < expt(2, p)
------------------------------------------------------------
{1}   expt(2, p) − 1 ≥ 0
{2}   binary_or(val′, binary_not(val′)) = expt(2, p) − 1
```

Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_or_not.2`.
Q.E.D.

C.16.28 Bitops.binary_xor_TCC1

Terse proof for `binary_xor_TCC1`.
`binary_xor_TCC1`:

```
------------------------------------------------------------
{1}   ∀ (v1, v2: below⌊expt(2, p)⌋):
        change_bits(v1, λ (n: nat): bit_set?(n, v1) ≠ bit_set?(n, v2))(p − 1) <
          expt(2, p)
```

Repeatedly Skolemizing and flattening,
Rewriting using change_bits_below, matching in *,
This completes the proof of `binary_xor_TCC1`.
Q.E.D.

C.16.29 Bitops.binary_xor_self

Terse proof for `binary_xor_self`.
`binary_xor_self`:

```
------------------------------------------------------------
{1}   ∀ (val: below⌊expt(2, p)⌋): binary_xor(val, val) = 0
```

Expanding the definition of binary_xor,
Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
we get 3 subgoals:
`binary_xor_self.1`:

```
{-1}  (∀ (n_1: nat):
         n_1 < p ⊃
           bit_set?(n_1, change_bits(val′, λ (n: nat): FALSE)(p − 1)) =
             bit_set?(n_1, 0))
        ≡ change_bits(val′, λ (n: nat): FALSE)(p − 1) = 0
{-2}  val′ < expt(2, p)
------------------------------------------------------------
{1}   change_bits(val′, λ (n: nat): FALSE)(p − 1) = 0
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Rewriting using bit_set_zero_mask, matching in *,
Rewriting using change_bits_result, matching in *,
This completes the proof of `binary_xor_self.1`.

`binary_xor_self.2`:

$$\begin{array}{ll}\{\text{-1}\} & \text{val}' < \text{expt}(2,\ p)\\ \hline \{1\} & \text{change\_bits}(\text{val}',\ \lambda\ (n\colon \text{nat})\colon \text{FALSE})(p-1) < \text{expt}(2,\ p)\\ \{2\} & \text{change\_bits}(\text{val}',\ \lambda\ (n\colon \text{nat})\colon \text{FALSE})(p-1) = 0\end{array}$$

Using lemma change_bits_below,
This completes the proof of `binary_xor_self.2`.
`binary_xor_self.3`:

$$\begin{array}{ll}\{\text{-1}\} & \text{val}' < \text{expt}(2,\ p)\\ \hline \{1\} & p-1 \geq 0\\ \{2\} & \text{change\_bits}(\text{val}',\ \lambda\ (n\colon \text{nat})\colon \text{FALSE})(p-1) = 0\end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_xor_self.3`.
Q.E.D.

C.16.30 Bitops.binary_or_full

Terse proof for `binary_or_full`.
`binary_or_full`:

$$\begin{array}{ll}\{1\} & \forall\ (\text{val}\colon \text{below}\lfloor\text{expt}(2,\ p)\rfloor)\colon\\ & \quad \text{binary\_or}(\text{val},\ \text{expt}(2,\ p)-1) = \text{expt}(2,\ p)-1\end{array}$$

Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
we get 2 subgoals:
`binary_or_full.1`:

$$\begin{array}{ll}\{\text{-1}\} & (\forall\ (n\colon \text{nat})\colon\\ & \quad n < p \supset\\ & \quad\ \text{bit\_set?}(n,\ \text{binary\_or}(\text{val}',\ \text{expt}(2,\ p)-1)) =\\ & \quad\ \text{bit\_set?}(n,\ \text{expt}(2,\ p)-1))\\ & \equiv \text{binary\_or}(\text{val}',\ \text{expt}(2,\ p)-1) = \text{expt}(2,\ p)-1\\ \{\text{-2}\} & \text{val}' < \text{expt}(2,\ p)\\ \hline \{1\} & \text{binary\_or}(\text{val}',\ \text{expt}(2,\ p)-1) = \text{expt}(2,\ p)-1\end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Expanding the definition of binary_or,
Hiding formulas: 2,
Rewriting using change_bits_result, matching in *,
we get 2 subgoals:
`binary_or_full.1.1`:

$$\begin{array}{ll}\{\text{-1}\} & n' \geq 0\\ \{\text{-2}\} & n' < p\\ \{\text{-3}\} & \text{val}' < \text{expt}(2,\ p)\\ \hline \{1\} & (\text{bit\_set?}(n',\ \text{val}') \vee \text{bit\_set?}(n',\ \text{expt}(2,\ p)-1)) =\\ & \quad \text{bit\_set?}(n',\ \text{expt}(2,\ p)-1)\end{array}$$

Rewriting using bit_set_full_mask, matching in *,
This completes the proof of `binary_or_full.1.1`.

`binary_or_full.1.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & n' \geq 0 \\
\{\text{-2}\} & n' < p \\
\{\text{-3}\} & \text{val}' < \text{expt}(2,\ p) \\
\hline
\{1\} & \forall\ (n\colon \text{nat})\colon \neg\ \text{bit\_set?}(n,\ \text{val}') \supset \text{expt}(2,\ p) - 1 \geq 0 \\
\{2\} & \text{bit\_set?}(n', \\
 & \qquad \text{change\_bits}(\text{val}', \\
 & \qquad\qquad \lambda\ (n\colon \text{nat})\colon \\
 & \qquad\qquad\quad \text{bit\_set?}(n,\ \text{val}')\ \vee\ \text{bit\_set?}(n,\ \text{expt}(2,\ p) - 1)) \\
 & \qquad\qquad (p - 1)) \\
 & \quad = \text{bit\_set?}(n',\ \text{expt}(2,\ p) - 1)
\end{array}
$$

Repeatedly Skolemizing and flattening,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_or_full.1.2`.
`binary_or_full.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & \text{val}' < \text{expt}(2,\ p) \\
\hline
\{1\} & \text{expt}(2,\ p) - 1 \geq 0 \\
\{2\} & \text{binary\_or}(\text{val}',\ \text{expt}(2,\ p) - 1) = \text{expt}(2,\ p) - 1
\end{array}
$$

Expanding the definition of binary_or,
Using lemma expt_ge1,
Expanding the definition of ^,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `binary_or_full.2`.
Q.E.D.

C.16.31 Bitops.binary_or_zero

Terse proof for `binary_or_zero`.
`binary_or_zero:`

$$
\begin{array}{ll}
 & \\
\hline
\{1\} & \forall\ (\text{val}\colon \text{below}\lfloor \text{expt}(2,\ p)\rfloor)\colon \text{binary\_or}(\text{val},\ 0) = \text{val}
\end{array}
$$

Repeatedly Skolemizing and flattening,
Using lemma bits_equal2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Expanding the definition of binary_or,
Rewriting using change_bits_result, matching in *,
Rewriting using bit_set_zero_mask, matching in *,
This completes the proof of `binary_or_zero`.
Q.E.D.

C.17 Proofs for Bits (bits.pvs)

C.17.1 Bits.bool_to_nat_below

Terse proof for `bool_to_nat_below`.

`bool_to_nat_below`:

$$\{1\} \quad \forall\ (b\colon \text{bool})\colon \text{bool\_to\_nat}(b) < 2$$

Repeatedly Skolemizing and flattening,
Expanding the definition of bool_to_nat,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bool_to_nat_below`.
Q.E.D.

C.17.2 Bits.nat_bool_iso

Terse proof for `nat_bool_iso`.
`nat_bool_iso`:

$$\{1\} \quad \forall\ (b\colon \text{bool})\colon \text{nat\_to\_bool}(\text{bool\_to\_nat}(b)) = b$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nat_bool_iso`.
Q.E.D.

C.17.3 Bits.cut_bit_zero

Terse proof for `cut_bit_zero`.
`cut_bit_zero`:

$$\{1\} \quad \forall\ (i\colon \text{nat})\colon \text{cut\_bit}(0,\ i) = \text{FALSE}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of cut_bit,
Rewriting using ndiv_0, matching in *,
Rewriting using rem_zero, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bit_zero`.
Q.E.D.

C.17.4 Bits.cut_bits_below

Terse proof for `cut_bits_below`.
`cut_bits_below`:

$$\{1\} \quad \forall\ (i,\ k,\ n\colon \text{nat})\colon \text{cut\_bits}(n,\ i,\ k) < \text{expt}(2,\ k)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cut_bits_below`.
Q.E.D.

C.17.5 Bits.cut_bits_aligned

Terse proof for `cut_bits_aligned`.

`cut_bits_aligned`:

{1} $\forall$ (n, i, k: nat): $i \leq k \wedge \text{cut\_bits}(n, 0, k) = 0 \supset \text{aligned?}(i)(n)$

Repeatedly Skolemizing and flattening,
Expanding the definition of cut_bits,
Expanding the definition of expt,
Rewriting using ndiv_1, matching in *,
Expanding the definition of aligned?,
Rewriting using rem_def2, matching in *,
Applying divides_expt_gt
Instantiating the top quantifier in -1 with the terms: n', k', i',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_aligned`.
Q.E.D.

C.17.6 Bits.aligned_cut_bits

Terse proof for `aligned_cut_bits`.
`aligned_cut_bits`:

{1} $\forall$ (n, i, j: nat, k: nat):
$i + j \leq k \wedge \text{aligned?}(k)(n) \supset \text{cut\_bits}(n, i, j) = 0$

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Expanding the definition of cut_bits,
Rewriting using rem_ndiv, matching in *,
Rewriting using expt_plus_aux, matching in *,
Applying rem_def2
Instantiating the top quantifier in -1 with the terms: expt(2, $i' + j'$), n', 0,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma divides_expt_gt,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_cut_bits`.
Q.E.D.

C.17.7 Bits.aligned_add_cut_bits

Terse proof for `aligned_add_cut_bits`.
`aligned_add_cut_bits`:

{1} $\forall$ (n, m, i, j, k: nat):
$i + j \leq k \wedge \text{aligned?}(k)(n) \supset$
$\text{cut\_bits}(n + m, i, j) = \text{cut\_bits}(m, i, j)$

Repeatedly Skolemizing and flattening,
Applying aligned_cut_bits
Instantiating the top quantifier in -1 with the terms: (n!1 i!1 j!1 k!1),
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of cut_bits,
Using lemma ndiv_plus_2,

Splitting conjunctions,
we get 2 subgoals:
`aligned_add_cut_bits.1:`

{-1} $\text{ndiv}(m' + n',\ \text{expt}(2,\ i')) =$
$\text{ndiv}(m',\ \text{expt}(2,\ i')) + \text{ndiv}(n',\ \text{expt}(2,\ i'))$
{-2} $\text{rem}(\text{expt}(2,\ j'))(\text{ndiv}(n',\ \text{expt}(2,\ i'))) = 0$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $i' \geq 0$
{-6} $j' \geq 0$
{-7} $k' \geq 0$
{-8} $i' + j' \leq k'$
{-9} $\text{aligned?}(k')(n')$

{1} $\text{rem}(\text{expt}(2,\ j'))(\text{ndiv}(m' + n',\ \text{expt}(2,\ i'))) =$
$\text{rem}(\text{expt}(2,\ j'))(\text{ndiv}(m',\ \text{expt}(2,\ i')))$

Replacing using formula -1,
Rewriting using rem_sum2, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_add_cut_bits.1`.
`aligned_add_cut_bits.2:`

{-1} $\text{rem}(\text{expt}(2,\ j'))(\text{ndiv}(n',\ \text{expt}(2,\ i'))) = 0$
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} $i' \geq 0$
{-5} $j' \geq 0$
{-6} $k' \geq 0$
{-7} $i' + j' \leq k'$
{-8} $\text{aligned?}(k')(n')$

{1} $\text{divides}(\text{expt}(2,\ i'),\ n')$
{2} $\text{rem}(\text{expt}(2,\ j'))(\text{ndiv}(m' + n',\ \text{expt}(2,\ i'))) =$
$\text{rem}(\text{expt}(2,\ j'))(\text{ndiv}(m',\ \text{expt}(2,\ i')))$

Hiding formulas: 2,
Expanding the definition of aligned?,
Rewriting using divides_transitive, matching in * where n gets expt(2, i!1), m gets expt(2, k!1), l gets n!1,
Hiding formulas: 2,
Expanding the definition of divides,
Instantiating the top quantifier in 1 with the terms: $\text{expt}(2, k' - i')$,
Keeping 1 and hiding *,
Rewriting using expt_plus_aux, matching in *,
This completes the proof of `aligned_add_cut_bits.2`.
Q.E.D.

C.17.8 Bits.cut_bit_bits

Terse proof for `cut_bit_bits`.
`cut_bit_bits:`

{1} $\forall\ (i,\ n\text{: nat})\text{: cut\_bit}(n,\ i) = \text{IF cut\_bits}(n,\ i,\ 1) = 1\ \text{THEN TRUE ELSE FALSE ENDIF}$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cut_bit_bits`.
Q.E.D.

C.17.9 Bits.cut_bits_zero

Terse proof for `cut_bits_zero`.
`cut_bits_zero:`

{1} $\forall$ (i, k: nat): cut_bits(0, i, k) = 0

Repeatedly Skolemizing and flattening,
Expanding the definition of cut_bits,
Installing automatic rewrites from: ndiv_0 rem_zero
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_zero`.
Q.E.D.

C.17.10 Bits.zero_cut_bits

Terse proof for `zero_cut_bits`.
`zero_cut_bits:`

{1} $\forall$ (n, i, k: nat): $k = 0 \supset$ cut_bits(n, i, k) = 0

Repeatedly Skolemizing and flattening,
Replacing using formula -4,
Expanding the definition of cut_bits,
Rewriting using expt_x0_aux, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `zero_cut_bits`.
Q.E.D.

C.17.11 Bits.cut_bits_cut_bits_TCC1

Terse proof for `cut_bits_cut_bits_TCC1`.
`cut_bits_cut_bits_TCC1:`

{1} $\forall$ (k_1, i_2, k_2: nat): $i_2 + k_2 > k_1 \wedge \neg\ i_2 \geq k_1 \supset k_1 - i_2 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cut_bits_cut_bits_TCC1`.
Q.E.D.

C.17.12 Bits.cut_bits_cut_bits

Terse proof for `cut_bits_cut_bits`.

`cut_bits_cut_bits:`

{1} $\forall\ (n,\ i_1,\ k_1,\ i_2,\ k_2\colon \text{nat})\colon$
cut_bits(cut_bits($n,\ i_1,\ k_1$), $i_2,\ k_2$) =
IF $i_2 \geq k_1$
THEN 0
ELSIF $i_2 + k_2 > k_1$ THEN cut_bits($n,\ i_1 + i_2,\ k_1 - i_2$)
ELSE cut_bits($n,\ i_1 + i_2,\ k_2$)
ENDIF

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`cut_bits_cut_bits.1:`

{-1} $n' \geq 0$
{-2} $i'_1 \geq 0$
{-3} $k'_1 \geq 0$
{-4} $i'_2 \geq 0$
{-5} $k'_2 \geq 0$
{-6} $i'_2 \geq k'_1$

{1} cut_bits(cut_bits($n',\ i'_1,\ k'_1$), $i'_2,\ k'_2$) = 0

Expanding the definition of cut_bits,

Applying ndiv_plus_mod

Installing automatic rewrites from: divides_zero ndiv_0 rem_zero

Instantiating the top quantifier in -1 with the terms: 0, expt(2, i'_2), rem(expt(2, k'_1))(ndiv(n', expt(2, i'_1))),

we get 2 subgoals:

`cut_bits_cut_bits.1.1:`

{-1} divides(expt(2, i'_2), 0) $\supset$
ndiv(0 + rem(expt(2, k'_1))(ndiv(n', expt(2, i'_1))), expt(2, i'_2)) =
ndiv(0, expt(2, i'_2))
{-2} $n' \geq 0$
{-3} $i'_1 \geq 0$
{-4} $k'_1 \geq 0$
{-5} $i'_2 \geq 0$
{-6} $k'_2 \geq 0$
{-7} $i'_2 \geq k'_1$

{1} rem(expt(2, k'_2))
(ndiv(rem(expt(2, k'_1))(ndiv(n', expt(2, i'_1))), expt(2, i'_2)))
= 0

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_cut_bits.1.1`.

`cut_bits_cut_bits.1.2:`

{-1} $n' \geq 0$
{-2} $i'_1 \geq 0$
{-3} $k'_1 \geq 0$
{-4} $i'_2 \geq 0$
{-5} $k'_2 \geq 0$
{-6} $i'_2 \geq k'_1$

{1} $\text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1))) < \text{expt}(2, i'_2)$
{2} $\text{rem}(\text{expt}(2, k'_2))$
$\quad(\text{ndiv}(\text{rem}(\text{expt}(2, k'_1))(\text{ndiv}(n', \text{expt}(2, i'_1))), \text{expt}(2, i'_2)))$
$\quad= 0$

Applying both_sides_expt_gt1_le
Instantiating the top quantifier in -1 with the terms: 2, k'_1, i'_2,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_cut_bits.1.2`.
`cut_bits_cut_bits.2:`

{-1} $n' \geq 0$
{-2} $i'_1 \geq 0$
{-3} $k'_1 \geq 0$
{-4} $i'_2 \geq 0$
{-5} $k'_2 \geq 0$
{-6} $i'_2 + k'_2 > k'_1$

{1} $i'_2 \geq k'_1$
{2} $\text{cut\_bits}(\text{cut\_bits}(n', i'_1, k'_1), i'_2, k'_2) =$
$\quad\text{cut\_bits}(n', i'_1 + i'_2, k'_1 - i'_2)$

Installing automatic rewrites from: cut_bits rem_ndiv expt_plus
Simplifying, rewriting, and recording with decision procedures,
Rewriting using ndiv_times_2, matching in *,
Applying rem_mod
Instantiating the top quantifier in -1 with the terms: $\text{expt}(2, i'_1 + i'_2 + k'_2)$, $\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')$,
we get 2 subgoals:
`cut_bits_cut_bits.2.1:`

{-1} $\text{rem}(\text{expt}(2, i'_1 + i'_2 + k'_2))(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')) =$
$\quad\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')$
{-2} $n' \geq 0$
{-3} $i'_1 \geq 0$
{-4} $k'_1 \geq 0$
{-5} $i'_2 \geq 0$
{-6} $k'_2 \geq 0$
{-7} $i'_2 + k'_2 > k'_1$

{1} $i'_2 \geq k'_1$
{2} $\text{ndiv}(\text{rem}(\text{expt}(2, i'_1 + i'_2 + k'_2))$
$\qquad(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n')),$
$\qquad\text{expt}(2, i'_1 + i'_2))$
$\quad= \text{ndiv}(\text{rem}(\text{expt}(2, i'_1 + k'_1))(n'), \text{expt}(2, i'_1 + i'_2))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_cut_bits.2.1`.

`cut_bits_cut_bits.2.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & n' \geq 0 \\
\{\text{-2}\} & i'_1 \geq 0 \\
\{\text{-3}\} & k'_1 \geq 0 \\
\{\text{-4}\} & i'_2 \geq 0 \\
\{\text{-5}\} & k'_2 \geq 0 \\
\{\text{-6}\} & i'_2 + k'_2 > k'_1 \\
\hline
\{1\} & \mathrm{rem}(\mathrm{expt}(2,\ i'_1 + k'_1))(n') < \mathrm{expt}(2,\ i'_1 + i'_2 + k'_2) \\
\{2\} & i'_2 \geq k'_1 \\
\{3\} & \mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ i'_1 + i'_2 + k'_2)) \\
 & \qquad\qquad (\mathrm{rem}(\mathrm{expt}(2,\ i'_1 + k'_1))(n')), \\
 & \qquad \mathrm{expt}(2,\ i'_1 + i'_2)) \\
 & \quad = \mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ i'_1 + k'_1))(n'),\ \mathrm{expt}(2,\ i'_1 + i'_2))
\end{array}
$$

Hiding formulas: 3,
Applying both_sides_expt_gt1_le
Instantiating the top quantifier in -1 with the terms: 2, $i'_1 + k'_1$, $i'_1 + i'_2 + k'_2$,
Simplifying, rewriting, and recording with decision procedures,
Adding type constraints for rem(expt(2, i1!1 + k1!1))(n!1),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_cut_bits.2.2`.
`cut_bits_cut_bits.3:`

$$
\begin{array}{ll}
\{\text{-1}\} & n' \geq 0 \\
\{\text{-2}\} & i'_1 \geq 0 \\
\{\text{-3}\} & k'_1 \geq 0 \\
\{\text{-4}\} & i'_2 \geq 0 \\
\{\text{-5}\} & k'_2 \geq 0 \\
\hline
\{1\} & i'_2 \geq k'_1 \\
\{2\} & i'_2 + k'_2 > k'_1 \\
\{3\} & \mathrm{cut\_bits}(\mathrm{cut\_bits}(n',\ i'_1,\ k'_1),\ i'_2,\ k'_2) = \mathrm{cut\_bits}(n',\ i'_1 + i'_2,\ k'_2)
\end{array}
$$

Installing automatic rewrites from: cut_bits rem_ndiv expt_plus
Simplifying, rewriting, and recording with decision procedures,
Rewriting using ndiv_times_2, matching in *,
Rewriting using rem_rem, matching in *,
Hiding formulas: 4,
Rewriting using expt_divides, matching in *,
This completes the proof of `cut_bits_cut_bits.3`.
Q.E.D.

C.17.13 Bits.cut_bit_cut_bits

Terse proof for `cut_bit_cut_bits`.
`cut_bit_cut_bits:`

$$
\begin{array}{ll}
 & \\
\hline
\{1\} & \forall\ (n,\ i_1,\ k_1,\ i_2\text{: nat}): \\
 & \quad \mathrm{cut\_bit}(\mathrm{cut\_bits}(n,\ i_1,\ k_1),\ i_2) = \\
 & \quad\ \text{IF } i_2 \geq k_1 \text{ THEN FALSE ELSE } \mathrm{cut\_bit}(n,\ i_1 + i_2) \text{ ENDIF}
\end{array}
$$

Repeatedly Skolemizing and flattening,
Rewriting using cut_bit_bits, matching in *,
Rewriting using cut_bit_bits, matching in *,

Rewriting using cut_bits_cut_bits, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cut_bit_cut_bits`.
Q.E.D.

C.17.14 Bits.shift_bits_left_less_TCC1

Terse proof for `shift_bits_left_less_TCC1`.
`shift_bits_left_less_TCC1:`

{1} $\forall\ (i,\ k\colon \text{nat})\colon\ k \geq i \supset k - i \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `shift_bits_left_less_TCC1`.
Q.E.D.

C.17.15 Bits.shift_bits_left_less

Terse proof for `shift_bits_left_less`.
`shift_bits_left_less:`

{1} $\forall\ (n,\ i,\ k\colon \text{nat})\colon$
$k \geq i \wedge n < \text{expt}(2,\ k - i) \supset \text{shift\_bits\_left}(n,\ i) < \text{expt}(2,\ k)$

Repeatedly Skolemizing and flattening,
Expanding the definition of shift_bits_left,
Applying both_sides_times_pos_lt1
Instantiating the top quantifier in -1 with the terms: expt(2, i'), n', expt(2, $k' - i'$),
we get 2 subgoals:
`shift_bits_left_less.1:`

{-1} $n' \times \text{expt}(2,\ i') < \text{expt}(2,\ k' - i') \times \text{expt}(2,\ i') \equiv n' < \text{expt}(2,\ k' - i')$
{-2} $n' \geq 0$
{-3} $i' \geq 0$
{-4} $k' \geq 0$
{-5} $k' \geq i'$
{-6} $n' < \text{expt}(2,\ k' - i')$

{1} $n' \times \text{expt}(2,\ i') < \text{expt}(2,\ k')$

Rewriting using expt_plus, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `shift_bits_left_less.1`.
`shift_bits_left_less.2:`

{-1} $n' \geq 0$
{-2} $i' \geq 0$
{-3} $k' \geq 0$
{-4} $k' \geq i'$
{-5} $n' < \text{expt}(2,\ k' - i')$

{1} $k' - i' \geq 0$
{2} $n' \times \text{expt}(2,\ i') < \text{expt}(2,\ k')$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `shift_bits_left_less.2`.
Q.E.D.

C.17.16 Bits.aligned_shift_bits_ok

Terse proof for `aligned_shift_bits_ok`.
`aligned_shift_bits_ok`:

{1} $\forall\ (n,\ i,\ j\colon \text{nat})\colon\ i \leq j \supset \text{aligned?}(i)(\text{shift\_bits\_left}(n,\ j))$

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Expanding the definition of shift_bits_left,
Rewriting using divides_prod2, matching in *,
Rewriting using expt_divides, matching in *,
This completes the proof of `aligned_shift_bits_ok`.
Q.E.D.

C.17.17 Bits.aligned_shift_bits_reduce_TCC1

Terse proof for `aligned_shift_bits_reduce_TCC1`.
`aligned_shift_bits_reduce_TCC1`:

{1} $\forall\ (i,\ j\colon \text{nat})\colon\ i > j \supset i - j \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `aligned_shift_bits_reduce_TCC1`.
Q.E.D.

C.17.18 Bits.aligned_shift_bits_reduce

Terse proof for `aligned_shift_bits_reduce`.
`aligned_shift_bits_reduce`:

{1} $\forall\ (n,\ i,\ j\colon \text{nat})\colon$ $\quad i > j \wedge \text{aligned?}(i-j)(n) \supset \text{aligned?}(i)(\text{shift\_bits\_left}(n,\ j))$

Repeatedly Skolemizing and flattening,
Expanding the definition of aligned?,
Expanding the definition of shift_bits_left,
Applying expt_plus
Instantiating the top quantifier in -1 with the terms: 2, $i' - j'$, j',
we get 2 subgoals:
`aligned_shift_bits_reduce.1`:

{-1} $\text{expt}(2,\ i' - j') \times \text{expt}(2,\ j') = \text{expt}(2,\ i' - j' + j')$
{-2} $n' \geq 0$
{-3} $i' \geq 0$
{-4} $j' \geq 0$
{-5} $i' > j'$
{-6} $\text{divides}(\text{expt}(2,\ i' - j'),\ n')$
{1} $\text{divides}(\text{expt}(2,\ i'),\ n' \times \text{expt}(2,\ j'))$

Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Using lemma divides_prod_elim2,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_shift_bits_reduce.1`.
`aligned_shift_bits_reduce.2`:

{-1} $n' \geq 0$
{-2} $i' \geq 0$
{-3} $j' \geq 0$
{-4} $i' > j'$
{-5} divides(expt(2, $i' - j'$), n')

{1} $i' - j' \geq 0$
{2} divides(expt(2, i'), $n' \times$ expt(2, j'))

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_shift_bits_reduce.2`.
Q.E.D.

C.17.19 Bits.overwrite_bits_TCC1

Terse proof for `overwrite_bits_TCC1`.
`overwrite_bits_TCC1`:

{1} $\forall$ (n, m, i, k: nat):
$n -$ rem(expt(2, $i + k$))(n) + shift_bits_left(cut_bits(m, i, k), i) + rem(expt(2, i))(n)
≥ 0

Repeatedly Skolemizing and flattening,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `overwrite_bits_TCC1`.
Q.E.D.

C.17.20 Bits.overwrite_bits_less

Terse proof for `overwrite_bits_less`.
`overwrite_bits_less`:

{1} $\forall$ (n, m, i, k, j: nat):
$n <$ expt(2, j) $\wedge$ $i + k \leq j \supset$ overwrite_bits(n, m, i, k) $<$ expt(2, j)

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_bits,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Applying bit_split_less
Instantiating the top quantifier in -1 with the terms: q', rem(expt(2, i'))(n') + shift_bits_left(cut_bits(m', i', k'), i'), $j' - i' - k'$, $i' + k'$,
Simplifying, rewriting, and recording with decision procedures,

Splitting conjunctions,
we get 2 subgoals:
`overwrite_bits_less.1:`

{-1} $q' \geq 0$
{-2} $n' = \text{rem}(\text{expt}(2,\ i' + k'))(n') + \text{expt}(2,\ i' + k') \times q'$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $i' \geq 0$
{-6} $k' \geq 0$
{-7} $j' \geq 0$
{-8} $n' < \text{expt}(2,\ j')$
{-9} $i' + k' \leq j'$

{1} $q' < \text{expt}(2,\ -1 \times i' - k' + j')$
{2} $\text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(m',\ i',\ k'),\ i') - \text{rem}(\text{expt}(2,\ i' + k'))(n') + n' < \text{expt}(2,\ j')$

Hiding formulas: 2,
Applying both_sides_times_pos_ge2
Instantiating the top quantifier in -1 with the terms: expt(2, $i' + k'$), q', expt(2, $-1 \times i' - k' + j'$),
Rewriting using expt_plus, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `overwrite_bits_less.1`.
`overwrite_bits_less.2:`

{-1} $q' \geq 0$
{-2} $n' = \text{rem}(\text{expt}(2,\ i' + k'))(n') + \text{expt}(2,\ i' + k') \times q'$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $i' \geq 0$
{-6} $k' \geq 0$
{-7} $j' \geq 0$
{-8} $n' < \text{expt}(2,\ j')$
{-9} $i' + k' \leq j'$

{1} $\text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(m',\ i',\ k'),\ i') < \text{expt}(2,\ i' + k')$
{2} $\text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(m',\ i',\ k'),\ i') - \text{rem}(\text{expt}(2,\ i' + k'))(n') + n' < \text{expt}(2,\ j')$

Hiding formulas: 2,
Expanding the definition of shift_bits_left,
Applying bit_split_less
Instantiating the top quantifier in -1 with the terms: cut_bits(m', i', k'), rem(expt(2, i'))(n'), k', i',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `overwrite_bits_less.2`.
Q.E.D.

C.17.21 Bits.overwrite_bit_less

Terse proof for `overwrite_bit_less`.
`overwrite_bit_less:`

{1} $\forall\ (n,\ m,\ i,\ k\text{: nat}):$
$n < \text{expt}(2,\ k) \wedge i < k \supset \text{overwrite\_bit}(n,\ m,\ i) < \text{expt}(2,\ k)$

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_bit,
Rewriting using overwrite_bits_less, matching in *,
This completes the proof of `overwrite_bit_less`.
Q.E.D.

C.17.22 Bits.overwrite_bool_bit_less

Terse proof for `overwrite_bool_bit_less`.
`overwrite_bool_bit_less:`

{1} $\forall$ (n: nat, b: bool, i, k: nat):
$n < \text{expt}(2, k) \wedge i < k \supset \text{overwrite\_bool\_bit}(n, b, i) < \text{expt}(2, k)$

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_bool_bit,
Rewriting using overwrite_bit_less, matching in *,
This completes the proof of `overwrite_bool_bit_less`.
Q.E.D.

C.17.23 Bits.overwrite_shift_bits_less

Terse proof for `overwrite_shift_bits_less`.
`overwrite_shift_bits_less:`

{1} $\forall$ (n, m, i, k, j: nat):
$n < \text{expt}(2, j) \wedge i + k \leq j \supset$
$\text{overwrite\_shift\_bits}(n, m, i, k) < \text{expt}(2, j)$

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_shift_bits,
Rewriting using overwrite_bits_less, matching in *,
This completes the proof of `overwrite_shift_bits_less`.
Q.E.D.

C.17.24 Bits.aligned_overwrite_shift_bits_more

Terse proof for `aligned_overwrite_shift_bits_more`.
`aligned_overwrite_shift_bits_more:`

{1} $\forall$ (a, n, m, i: nat, k: posnat):
$a \leq i \wedge \text{aligned?}(a)(n) \supset \text{aligned?}(a)(\text{overwrite\_shift\_bits}(n, m, i, k))$

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_shift_bits,
Expanding the definition of overwrite_bits,
Applying aligned_plus_n
Instantiating the top quantifier in -1 with the terms: a', $n' - \text{rem}(\text{expt}(2, i' + k'))(n')$, $\text{rem}(\text{expt}(2, i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(\text{shift\_bits\_left}(m', i'), i', k'), i')$,
we get 3 subgoals:

`aligned_overwrite_shift_bits_more.1:`

{-1}	aligned?(a') (n' − rem(expt(2, $i' + k'$))(n') + rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_
{-2}	$a' \geq 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$i' \geq 0$
{-6}	$k' > 0$
{-7}	$a' \leq i'$
{-8}	aligned?(a')(n')
{1}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i') − rem(e

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_more.1`.

`aligned_overwrite_shift_bits_more.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?(a')(n')
{1}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))
{2}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i') − rem(e

Hiding formulas: 2,

Rewriting using aligned_plus_n, matching in *,

we get 2 subgoals:

`aligned_overwrite_shift_bits_more.2.1:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?(a')(n')
{1}	aligned?(a')(shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))
{2}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))

Hiding formulas: 2,

Rewriting using aligned_shift_bits_ok, matching in *,

This completes the proof of `aligned_overwrite_shift_bits_more.2.1`.

`aligned_overwrite_shift_bits_more.2.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?$(a')(n')$
{1}	aligned?$(a')(\text{rem}(\text{expt}(2,\ i'))(n'))$
{2}	aligned?(a') $(\text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(\text{shift\_bits\_left}(m',\ i'),\ i',\ k'),\ i'))$

Rewriting using aligned_rem, matching in *,

This completes the proof of `aligned_overwrite_shift_bits_more.2.2`.

`aligned_overwrite_shift_bits_more.3:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?$(a')(n')$
{1}	$n' - \text{rem}(\text{expt}(2,\ i' + k'))(n') \geq 0 \wedge$ aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$
{2}	aligned?(a') $(\text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(\text{shift\_bits\_left}(m',\ i'),\ i',\ k'),\ i') - \text{rem}(\text{expt}(2,\ i' +$

Hiding formulas: 2,

Applying propositional simplification,

we get 2 subgoals:

`aligned_overwrite_shift_bits_more.3.1:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?$(a')(n')$
{1}	$n' - \text{rem}(\text{expt}(2,\ i' + k'))(n') \geq 0$

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_more.3.1`.

`aligned_overwrite_shift_bits_more.3.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?$(a')(n')$
{1}	aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$

Applying aligned_minus
Instantiating the top quantifier in -1 with the terms: a', a', n', rem(expt(2, $i' + k'$))(n'),
we get 2 subgoals:
`aligned_overwrite_shift_bits_more.3.2.1:`

{-1}	$n' \geq \text{rem}(\text{expt}(2,\ i' + k'))(n') \supset$ aligned?$(\min(a',\ a'))(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$
{-2}	$a' \geq 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$i' \geq 0$
{-6}	$k' > 0$
{-7}	$a' \leq i'$
{-8}	aligned?$(a')(n')$
{1}	aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$

Expanding the definition of min,
Simplifying, rewriting, and recording with decision procedures,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_overwrite_shift_bits_more.3.2.1`.
`aligned_overwrite_shift_bits_more.3.2.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$a' \leq i'$
{-7}	aligned?$(a')(n')$
{1}	aligned?$(a')(\text{rem}(\text{expt}(2,\ i' + k'))(n'))$
{2}	aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$

Rewriting using aligned_rem, matching in *,
This completes the proof of `aligned_overwrite_shift_bits_more.3.2.2`.
Q.E.D.

C.17.25 Bits.aligned_overwrite_shift_bits_less_TCC1

Terse proof for `aligned_overwrite_shift_bits_less_TCC1`.

`aligned_overwrite_shift_bits_less_TCC1:`

{1} $\forall$ (a, n, i: nat): aligned?(a)(n) $\wedge$ $i < a \supset a - i \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `aligned_overwrite_shift_bits_less_TCC1`.

Q.E.D.

C.17.26 Bits.aligned_overwrite_shift_bits_less

Terse proof for `aligned_overwrite_shift_bits_less`.

`aligned_overwrite_shift_bits_less:`

{1} $\forall$ (a, n, m, i: nat, k: posnat):
$i < a \wedge$ aligned?(a)(n) $\wedge$ aligned?($a - i$)(m) $\supset$
aligned?(a)(overwrite_shift_bits(n, m, i, k))

Installing automatic rewrites from: aligned_shift_bits_reduce ndiv_times_divident_2 ndiv_self aligned_rem divides_reflexive

Repeatedly Skolemizing and flattening,

Expanding the definition of overwrite_shift_bits,

Expanding the definition of overwrite_bits,

Applying aligned_plus_n

Instantiating the top quantifier in -1 with the terms: a', $n' - \text{rem}(\text{expt}(2, i' + k'))(n')$, $\text{rem}(\text{expt}(2, i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(\text{shift\_bits\_left}(m', i'), i', k'), i')$,

we get 3 subgoals:

`aligned_overwrite_shift_bits_less.1:`

{-1} aligned?(a')
($n' - \text{rem}(\text{expt}(2,\ i' + k'))(n') + \text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(\text{shift\_bits\_left}(m',\ i')$
{-2} $a' \geq 0$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $i' \geq 0$
{-6} $k' > 0$
{-7} $i' < a'$
{-8} aligned?(a')(n')
{-9} aligned?($a' - i'$)(m')

{1} aligned?(a')
($\text{rem}(\text{expt}(2,\ i'))(n') + \text{shift\_bits\_left}(\text{cut\_bits}(\text{shift\_bits\_left}(m',\ i'),\ i',\ k'),\ i') - \text{rem}(\text{expt}(2,\ i' +$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_less.1`.

`aligned_overwrite_shift_bits_less.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))
{2}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i') − rem(e

Hiding formulas: 2,
Rewriting using aligned_plus_n, matching in *,
we get 2 subgoals:
`aligned_overwrite_shift_bits_less.2.1:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	aligned?(a')(shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))
{2}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))

Hiding formulas: 2,
Rewriting using aligned_shift_bits_reduce, matching in *,
Hiding formulas: 2,
Expanding the definition of cut_bits,
Expanding the definition of shift_bits_left,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_overwrite_shift_bits_less.2.1`.
`aligned_overwrite_shift_bits_less.2.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	aligned?(a')(rem(expt(2, i'))(n'))
{2}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i'))

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_overwrite_shift_bits_less.2.2`.

`aligned_overwrite_shift_bits_less.3:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	$n' - \text{rem}(\text{expt}(2,\ i' + k'))(n') \geq 0 \wedge$ aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$
{2}	aligned?(a') (rem(expt(2, i'))(n') + shift_bits_left(cut_bits(shift_bits_left(m', i'), i', k'), i') − rem(expt(2, i' +

Hiding formulas: 2,

Applying propositional simplification,

we get 2 subgoals:

`aligned_overwrite_shift_bits_less.3.1:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	$n' - \text{rem}(\text{expt}(2,\ i' + k'))(n') \geq 0$

Using lemma rem_def_pos,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `aligned_overwrite_shift_bits_less.3.1`.

`aligned_overwrite_shift_bits_less.3.2:`

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i' + k'))(n'))$

Applying aligned_minus

Instantiating the top quantifier in -1 with the terms: a', a', n', rem(expt(2, $i' + k'$))(n'),

we get 2 subgoals:

`aligned_overwrite_shift_bits_less.3.2.1`:

{-1}	$n' \geq \text{rem}(\text{expt}(2,\ i'+k'))(n') \supset$ aligned?$(\min(a',\ a'))(n' - \text{rem}(\text{expt}(2,\ i'+k'))(n'))$
{-2}	$a' \geq 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$i' \geq 0$
{-6}	$k' > 0$
{-7}	$i' < a'$
{-8}	aligned?$(a')(n')$
{-9}	aligned?$(a' - i')(m')$
{1}	aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i'+k'))(n'))$

Expanding the definition of min,
Simplifying, rewriting, and recording with decision procedures,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_overwrite_shift_bits_less.3.2.1`.
`aligned_overwrite_shift_bits_less.3.2.2`:

{-1}	$a' \geq 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' > 0$
{-6}	$i' < a'$
{-7}	aligned?$(a')(n')$
{-8}	aligned?$(a' - i')(m')$
{1}	aligned?$(a')(\text{rem}(\text{expt}(2,\ i'+k'))(n'))$
{2}	aligned?$(a')(n' - \text{rem}(\text{expt}(2,\ i'+k'))(n'))$

Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `aligned_overwrite_shift_bits_less.3.2.2`.
Q.E.D.

C.17.27 Bits.cut_bits_shift_bits_left_TCC1

Terse proof for `cut_bits_shift_bits_left_TCC1`.
`cut_bits_shift_bits_left_TCC1`:

{1}	$\forall$ (si, ci, ck: nat): ci $<$ si $\wedge$ $\neg$ ci + ck $\leq$ si $\supset$ si $-$ ci $\geq$ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cut_bits_shift_bits_left_TCC1`.
Q.E.D.

C.17.28 Bits.cut_bits_shift_bits_left_TCC2

Terse proof for `cut_bits_shift_bits_left_TCC2`.

cut_bits_shift_bits_left_TCC2:

{1} ∀ (si, ci, ck: nat): ¬ ci < si ∧ ¬ ci + ck ≤ si ⊃ ci − si ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of cut_bits_shift_bits_left_TCC2.
Q.E.D.

C.17.29 Bits.cut_bits_shift_bits_left

Terse proof for cut_bits_shift_bits_left.
cut_bits_shift_bits_left:

{1} ∀ (n, si, ci, ck: nat):
cut_bits(shift_bits_left(n, si), ci, ck) =
IF ci + ck ≤ si
THEN 0
ELSIF ci < si THEN cut_bits(shift_bits_left(n, si − ci), 0, ck)
ELSE cut_bits(n, ci − si, ck)
ENDIF

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
cut_bits_shift_bits_left.1:

{-1} $n' \geq 0$
{-2} $si' \geq 0$
{-3} $ci' \geq 0$
{-4} $ck' \geq 0$
{-5} $ci' + ck' \leq si'$
{1} cut_bits(shift_bits_left(n', si'), ci', ck') = 0

Expanding the definition of cut_bits,
Rewriting using rem_def2, matching in *,
Expanding the definition of shift_bits_left,
Installing automatic rewrites from: ndiv_times_divident_2 expt_divides ndiv_expt_expt
Simplifying, rewriting, and recording with decision procedures,
Rewriting using divides_prod2, matching in *,
This completes the proof of cut_bits_shift_bits_left.1.
cut_bits_shift_bits_left.2:

{-1} $n' \geq 0$
{-2} $si' \geq 0$
{-3} $ci' \geq 0$
{-4} $ck' \geq 0$
{-5} $ci' < si'$
{1} $ci' + ck' \leq si'$
{2} cut_bits(shift_bits_left(n', si'), ci', ck') =
cut_bits(shift_bits_left(n', $si' - ci'$), 0, ck')

Expanding the definition of shift_bits_left,
Expanding the definition of cut_bits,

Installing automatic rewrites from: ndiv_times_divident_2 expt_divides ndiv_expt_expt expt_x0_aux ndiv_1

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_shift_bits_left.2`.

`cut_bits_shift_bits_left.3:`

$$
\begin{array}{ll}
\{-1\} & n' \geq 0 \\
\{-2\} & \mathrm{si}' \geq 0 \\
\{-3\} & \mathrm{ci}' \geq 0 \\
\{-4\} & \mathrm{ck}' \geq 0 \\
\hline
\{1\} & \mathrm{ci}' + \mathrm{ck}' \leq \mathrm{si}' \\
\{2\} & \mathrm{ci}' < \mathrm{si}' \\
\{3\} & \mathrm{cut\_bits}(\mathrm{shift\_bits\_left}(n', \mathrm{si}'), \mathrm{ci}', \mathrm{ck}') = \mathrm{cut\_bits}(n', \mathrm{ci}' - \mathrm{si}', \mathrm{ck}')
\end{array}
$$

Expanding the definition of shift_bits_left,

Expanding the definition of cut_bits,

Case splitting on ndiv(n!1 * expt(2, si!1), expt(2, ci!1)) = ndiv(n!1, expt(2, ci!1 - si!1)),

we get 2 subgoals:

`cut_bits_shift_bits_left.3.1:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{ndiv}(n' \times \mathrm{expt}(2, \mathrm{si}'), \mathrm{expt}(2, \mathrm{ci}')) = \\
 & \quad \mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}' - \mathrm{si}')) \\
\{-2\} & n' \geq 0 \\
\{-3\} & \mathrm{si}' \geq 0 \\
\{-4\} & \mathrm{ci}' \geq 0 \\
\{-5\} & \mathrm{ck}' \geq 0 \\
\hline
\{1\} & \mathrm{ci}' + \mathrm{ck}' \leq \mathrm{si}' \\
\{2\} & \mathrm{ci}' < \mathrm{si}' \\
\{3\} & \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n' \times \mathrm{expt}(2, \mathrm{si}'), \mathrm{expt}(2, \mathrm{ci}'))) = \\
 & \quad \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}' - \mathrm{si}')))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_shift_bits_left.3.1`.

`cut_bits_shift_bits_left.3.2:`

$$
\begin{array}{ll}
\{-1\} & n' \geq 0 \\
\{-2\} & \mathrm{si}' \geq 0 \\
\{-3\} & \mathrm{ci}' \geq 0 \\
\{-4\} & \mathrm{ck}' \geq 0 \\
\hline
\{1\} & \mathrm{ndiv}(n' \times \mathrm{expt}(2, \mathrm{si}'), \mathrm{expt}(2, \mathrm{ci}')) = \\
 & \quad \mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}' - \mathrm{si}')) \\
\{2\} & \mathrm{ci}' + \mathrm{ck}' \leq \mathrm{si}' \\
\{3\} & \mathrm{ci}' < \mathrm{si}' \\
\{4\} & \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n' \times \mathrm{expt}(2, \mathrm{si}'), \mathrm{expt}(2, \mathrm{ci}'))) = \\
 & \quad \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}' - \mathrm{si}')))
\end{array}
$$

Installing automatic rewrites from: expt_plus

Rewriting using ndiv_reduce, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_shift_bits_left.3.2`.

Q.E.D.

C.17.30 Bits.cut_bits_shift_bits_left_outside

Terse proof for `cut_bits_shift_bits_left_outside`.

cut_bits_shift_bits_left_outside:

{1} ∀ (n, si, ci, ck: nat):
ci + ck ≤ si ⊃ cut_bits(shift_bits_left(n, si), ci, ck) = 0

Repeatedly Skolemizing and flattening,
Rewriting using cut_bits_shift_bits_left, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of cut_bits_shift_bits_left_outside.
Q.E.D.

C.17.31 Bits.cut_bits_shift_bits_left_overlap_TCC1

Terse proof for cut_bits_shift_bits_left_overlap_TCC1.
cut_bits_shift_bits_left_overlap_TCC1:

{1} ∀ (si, ci, ck: nat): ci + ck > si ∧ ci < si ∧ ci > 0 ⊃ si − ci ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of cut_bits_shift_bits_left_overlap_TCC1.
Q.E.D.

C.17.32 Bits.cut_bits_shift_bits_left_overlap

Terse proof for cut_bits_shift_bits_left_overlap.
cut_bits_shift_bits_left_overlap:

{1} ∀ (n, si, ci, ck: nat):
ci + ck > si ∧ ci < si ∧ ci > 0 ⊃
cut_bits(shift_bits_left(n, si), ci, ck) =
cut_bits(shift_bits_left(n, si − ci), 0, ck)

Repeatedly Skolemizing and flattening,
Rewriting using cut_bits_shift_bits_left, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of cut_bits_shift_bits_left_overlap.
Q.E.D.

C.17.33 Bits.cut_bits_shift_bits_left_in

Terse proof for cut_bits_shift_bits_left_in.
cut_bits_shift_bits_left_in:

{1} ∀ (n, si, ci, ck: nat):
ci ≥ si ⊃ cut_bits(shift_bits_left(n, si), ci, ck) = cut_bits(n, ci − si, ck)

Repeatedly Skolemizing and flattening,
Rewriting using cut_bits_shift_bits_left, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Case splitting on ck!1 = 0,
we get 2 subgoals:

`cut_bits_shift_bits_left_in.1:`

{-1}	$\mathrm{ck}' = 0$
{-2}	$n' \geq 0$
{-3}	$\mathrm{si}' \geq 0$
{-4}	$\mathrm{ci}' \geq 0$
{-5}	$\mathrm{ck}' \geq 0$
{-6}	$\mathrm{ci}' \geq \mathrm{si}'$
{-7}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{si}'$
{1}	$0 = \mathrm{cut\_bits}(n', \mathrm{ci}' - \mathrm{si}', \mathrm{ck}')$

Expanding the definition of cut_bits,
Replacing using formula -1,
Rewriting using expt_x0_aux, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_shift_bits_left_in.1`.
`cut_bits_shift_bits_left_in.2:`

{-1}	$n' \geq 0$
{-2}	$\mathrm{si}' \geq 0$
{-3}	$\mathrm{ci}' \geq 0$
{-4}	$\mathrm{ck}' \geq 0$
{-5}	$\mathrm{ci}' \geq \mathrm{si}'$
{-6}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{si}'$
{1}	$\mathrm{ck}' = 0$
{2}	$0 = \mathrm{cut\_bits}(n', \mathrm{ci}' - \mathrm{si}', \mathrm{ck}')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_shift_bits_left_in.2`.
Q.E.D.

C.17.34 Bits.overwrite_shift_bits_zero

Terse proof for `overwrite_shift_bits_zero`.
`overwrite_shift_bits_zero:`

{1}	$\forall\ (z, m, i, k\colon \mathrm{nat}):$ $z = 0 \supset$ $\mathrm{overwrite\_shift\_bits}(z, m, i, k) = \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m, 0, k), i)$

Repeatedly Skolemizing and flattening,
Replacing using formula -5,
Expanding the definition of overwrite_shift_bits,
Expanding the definition of overwrite_bits,
Rewriting using rem_zero, matching in *,
Rewriting using rem_zero, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using cut_bits_shift_bits_left, matching in *,
Case splitting on k!1 = 0,
we get 2 subgoals:

`overwrite_shift_bits_zero.1:`

{-1}	$k' = 0$
{-2}	$0 \geq 0$
{-3}	$m' \geq 0$
{-4}	$i' \geq 0$
{-5}	$k' \geq 0$
{-6}	$z' = 0$
{1}	shift_bits_left(IF $i' + k' \leq i'$ THEN 0 ELSE cut_bits(m', 0, k') ENDIF, i') = shift_bits_left(cut_bits(m', 0, k'), i')

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using zero_cut_bits, matching in *,
This completes the proof of `overwrite_shift_bits_zero.1`.
`overwrite_shift_bits_zero.2:`

{-1}	$0 \geq 0$
{-2}	$m' \geq 0$
{-3}	$i' \geq 0$
{-4}	$k' \geq 0$
{-5}	$z' = 0$
{1}	$k' = 0$
{2}	shift_bits_left(IF $i' + k' \leq i'$ THEN 0 ELSE cut_bits(m', 0, k') ENDIF, i') = shift_bits_left(cut_bits(m', 0, k'), i')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `overwrite_shift_bits_zero.2`.
Q.E.D.

C.17.35 Bits.zero_overwrite_shift_bits

Terse proof for `zero_overwrite_shift_bits`.
`zero_overwrite_shift_bits:`

{1}	$\forall$ (n, m, i, k: nat): $k = 0 \supset$ overwrite_shift_bits(n, m, i, k) = n

Repeatedly Skolemizing and flattening,
Replacing using formula -5,
Hiding formulas: -4, -5,
Expanding the definition of overwrite_shift_bits,
Expanding the definition of overwrite_bits,
Rewriting using zero_cut_bits, matching in *,
Expanding the definition of shift_bits_left,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `zero_overwrite_shift_bits`.
Q.E.D.

C.17.36 Bits.bit_split_TCC1

Terse proof for `bit_split_TCC1`.
`bit_split_TCC1:`

{1}	$\forall$ (n, i, k: nat): $k \leq i \wedge n <$ expt(2, i) $\supset i - k \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `bit_split_TCC1`.
Q.E.D.

C.17.37 Bits.bit_split

Terse proof for `bit_split`.
`bit_split:`

{1}	$\forall\ (n,\ i,\ k\text{: nat}):$ $k \leq i \wedge n < \text{expt}(2,\ i) \supset$ $n =$ $\text{overwrite\_shift\_bits}(\text{cut\_bits}(n,\ 0,\ k),\ \text{cut\_bits}(n,\ k,\ i-k),\ k,\ i-k)$

Repeatedly Skolemizing and flattening,
Case splitting on k!1 = i!1,
we get 2 subgoals:
`bit_split.1:`

{-1}	$k' = i'$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	$k' \geq 0$
{-5}	$k' \leq i'$
{-6}	$n' < \text{expt}(2,\ i')$
{1}	$n' =$ $\text{overwrite\_shift\_bits}(\text{cut\_bits}(n',\ 0,\ k'),\ \text{cut\_bits}(n',\ k',\ i'-k'),\ k',\ i'-k')$

Simplifying, rewriting, and recording with decision procedures,
Applying zero_cut_bits
Instantiating the top quantifier in -1 with the terms: n', k', $i' - k'$,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Rewriting using zero_overwrite_shift_bits, matching in *,
Expanding the definition of cut_bits,
Rewriting using rem_mod, matching in *,
Rewriting using expt_x0_aux, matching in *,
Rewriting using ndiv_1, matching in *,
This completes the proof of `bit_split.1`.
`bit_split.2:`

{-1}	$n' \geq 0$
{-2}	$i' \geq 0$
{-3}	$k' \geq 0$
{-4}	$k' \leq i'$
{-5}	$n' < \text{expt}(2,\ i')$
{1}	$k' = i'$
{2}	$n' =$ $\text{overwrite\_shift\_bits}(\text{cut\_bits}(n',\ 0,\ k'),\ \text{cut\_bits}(n',\ k',\ i'-k'),\ k',\ i'-k')$

Expanding the definition of overwrite_shift_bits,
Expanding the definition of overwrite_bits,
Rewriting using cut_bits_shift_bits_left, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using cut_bits_cut_bits, matching in *,
Rewriting using rem_mod, matching in *,
Rewriting using rem_mod, matching in *,
we get 2 subgoals:
`bit_split.2.1:`

{-1} $n' \geq 0$
{-2} $i' \geq 0$
{-3} $k' \geq 0$
{-4} $k' \leq i'$
{-5} $n' < \text{expt}(2,\ i')$

{1} $k' = i'$
{2} $n' =$
$\text{cut\_bits}(n',\ 0,\ k') + \text{shift\_bits\_left}(\text{cut\_bits}(n',\ k',\ i' - k'),\ k') + \text{cut\_bits}(n',\ 0,\ k') - \text{cut\_bits}(n',\ 0,\ k')$

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of shift_bits_left,
Expanding the definition of cut_bits,
Rewriting using expt_x0_aux, matching in *,
Rewriting using ndiv_1, matching in *,
Rewriting using rem_ndiv, matching in *,
Rewriting using expt_plus, matching in *,
Applying rem_mod
Instantiating the top quantifier in -1 with the terms: expt(2, i'), n',
Replacing using formula -1,
Hiding formulas: -1,
Using lemma number_split,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `bit_split.2.1`.
`bit_split.2.2:`

{-1} $n' \geq 0$
{-2} $i' \geq 0$
{-3} $k' \geq 0$
{-4} $k' \leq i'$
{-5} $n' < \text{expt}(2,\ i')$

{1} $\text{cut\_bits}(n',\ 0,\ k') < \text{expt}(2,\ i')$
{2} $k' = i'$
{3} $n' =$
$\text{shift\_bits\_left}(\text{cut\_bits}(n',\ k',\ i' - k'),\ k') + 2 \times \text{cut\_bits}(n',\ 0,\ k') - \text{rem}(\text{expt}(2,\ i'))(\text{cut\_bits}(n',\ 0,\ k'))$

Hiding formulas: 3,
Using lemma both_sides_expt_gt1_le,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `bit_split.2.2`.
Q.E.D.

C.17.38 Bits.shift_bits_left_cut_bits

Terse proof for `shift_bits_left_cut_bits`.

`shift_bits_left_cut_bits:`

{1}	$\forall$ (n, i, k, j: nat): $i = j \wedge$ aligned?(i)(n) $\wedge$ $n <$ expt(2, $i+k$) $\supset$ shift_bits_left(cut_bits(n, i, k), j) $= n$

Repeatedly Skolemizing and flattening,
Case splitting on k!1 = 0,
we get 2 subgoals:
`shift_bits_left_cut_bits.1:`

{-1}	$k' = 0$
{-2}	$n' \geq 0$
{-3}	$i' \geq 0$
{-4}	$k' \geq 0$
{-5}	$j' \geq 0$
{-6}	$i' = j'$
{-7}	aligned?(i')(n')
{-8}	$n' <$ expt(2, $i'+k'$)
{1}	shift_bits_left(cut_bits(n', i', k'), j') $= n'$

Replacing using formula -1,
Rewriting using zero_cut_bits, matching in *,
Expanding the definition of shift_bits_left,
Rewriting using zero_times1, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of aligned?,
Applying rem_mod
Instantiating the top quantifier in -1 with the terms: expt(2, i'), n',
Applying rem_def2
Instantiating the top quantifier in -1 with the terms: expt(2, i'), n', 0,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `shift_bits_left_cut_bits.1`.
`shift_bits_left_cut_bits.2:`

{-1}	$n' \geq 0$
{-2}	$i' \geq 0$
{-3}	$k' \geq 0$
{-4}	$j' \geq 0$
{-5}	$i' = j'$
{-6}	aligned?(i')(n')
{-7}	$n' <$ expt(2, $i'+k'$)
{1}	$k' = 0$
{2}	shift_bits_left(cut_bits(n', i', k'), j') $= n'$

Using lemma bit_split,
Simplifying, rewriting, and recording with decision procedures,
Using lemma aligned_cut_bits,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Rewriting using overwrite_shift_bits_zero, matching in *,
Rewriting using cut_bits_cut_bits, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `shift_bits_left_cut_bits.2`.

Q.E.D.

C.17.39 Bits.cut_bits_0_to_below

Terse proof for `cut_bits_0_to_below`.
`cut_bits_0_to_below:`

{1} $\forall$ (n, k: nat): $n < \text{expt}(2, k) \supset \text{cut\_bits}(n, 0, k) = n$

Repeatedly Skolemizing and flattening,
Applying shift_bits_left_cut_bits
Instantiating the top quantifier in -1 with the terms: n', 0, k', 0,
Rewriting using zero_aligned, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of shift_bits_left,
Rewriting using expt_x0_aux, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_0_to_below`.
Q.E.D.

C.17.40 Bits.cut_bits_overwrite_bits_disjoined

Terse proof for `cut_bits_overwrite_bits_disjoined`.
`cut_bits_overwrite_bits_disjoined:`

{1} $\forall$ (n, m, oi, ok: nat, ci, ck: nat):
oi + ok $\leq$ ci $\vee$ ci + ck $\leq$ oi $\supset$
cut_bits(overwrite_bits(n, m, oi, ok), ci, ck) = cut_bits(n, ci, ck)

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_bits,
Expanding the definition of cut_bits,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`cut_bits_overwrite_bits_disjoined.1:`

{-1} $n' \geq 0$
{-2} $m' \geq 0$
{-3} oi$'$ ≥ 0
{-4} ok$'$ ≥ 0
{-5} ci$'$ ≥ 0
{-6} ck$'$ ≥ 0
{-7} oi$'$ + ok$'$ $\leq$ ci$'$

{1} rem(expt(2, ck$'$))
(ndiv(rem(expt(2, oi$'$))(n') + shift_bits_left(rem(expt(2, ok$'$))(ndiv(m', expt(2, oi$'$))), oi$'$) − rem(expt(2,
expt(2, ci$'$)))
= rem(expt(2, ck$'$))(ndiv(n', expt(2, ci$'$)))

Case splitting on ndiv(rem(expt(2, oi!1))(n!1) + shift_bits_left(rem(expt(2, ok!1))(ndiv(m!1, expt(2, oi!1))), oi!1) - rem(expt(2, oi!1 + ok!1))(n!1) + n!1, expt(2, ci!1)) = ndiv(rem(expt(2, oi!1))(n!1) + shift_bits_left(rem(expt(2, ok!1))(ndiv(m!1, expt(2, oi!1))), oi!1) - rem(expt(2, oi!1 + ok!1))(n!1) + n!1, expt(2, oi!1 + ok!1) * expt(2, ci!1 - oi!1 - ok!1)),
we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.1.1:`

{-1} $\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))),\ \mathrm{oi}') - \mathrm{rem}($
$\mathrm{expt}(2,\ \mathrm{ci}'))$
$=$
$\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))),\ \mathrm{oi}') - \mathrm{rem}$
$\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}') \times \mathrm{expt}(2,\ \mathrm{ci}' - \mathrm{oi}' - \mathrm{ok}'))$

{-2} $n' \geq 0$

{-3} $m' \geq 0$

{-4} $\mathrm{oi}' \geq 0$

{-5} $\mathrm{ok}' \geq 0$

{-6} $\mathrm{ci}' \geq 0$

{-7} $\mathrm{ck}' \geq 0$

{-8} $\mathrm{oi}' + \mathrm{ok}' \leq \mathrm{ci}'$

{1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))$
$(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))),\ \mathrm{oi}') -$
$\mathrm{expt}(2,\ \mathrm{ci}')))$
$=\ \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(n',\ \mathrm{expt}(2,\ \mathrm{ci}')))$

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using ndiv_times_2, matching in *,

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: $\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')$, n', $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying ndiv_plus_mod_2

Instantiating the top quantifier in -1 with the terms: $n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n')$, $\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')$, $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}')$, $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n')$,

we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.1.1.1:`

{-1} $\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n')) \supset$
$\mathrm{ndiv}(n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n') + \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(r$
$\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))$
$=$
$\mathrm{ndiv}(n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n') + \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n'),$
$\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))$

{-2} $\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n'))$

{-3} $n' \geq 0$

{-4} $m' \geq 0$

{-5} $\mathrm{oi}' \geq 0$

{-6} $\mathrm{ok}' \geq 0$

{-7} $\mathrm{ci}' \geq 0$

{-8} $\mathrm{ck}' \geq 0$

{-9} $\mathrm{oi}' + \mathrm{ok}' \leq \mathrm{ci}'$

{1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))$
$(\mathrm{ndiv}(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))),\ c$
$\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')),$
$\mathrm{expt}(2,\ -1 \times \mathrm{oi}' - \mathrm{ok}' + \mathrm{ci}')))$
$=\ \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(n',\ \mathrm{expt}(2,\ \mathrm{ci}')))$

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using ndiv_times_2, matching in *,

Rewriting using expt_plus, matching in *,

This completes the proof of `cut_bits_overwrite_bits_disjoined.1.1.1`.

`cut_bits_overwrite_bits_disjoined.1.1.2`:

{-1}	divides(expt(2, $\mathrm{oi}' + \mathrm{ok}'$), $n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n')$)
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$\mathrm{oi}' \geq 0$
{-5}	$\mathrm{ok}' \geq 0$
{-6}	$\mathrm{ci}' \geq 0$
{-7}	$\mathrm{ck}' \geq 0$
{-8}	$\mathrm{oi}' + \mathrm{ok}' \leq \mathrm{ci}'$
{1}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}')$ $< \mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')$
{2}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') - \mathrm{rem}(\mathrm{ex}$ $\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')),$ $\mathrm{expt}(2, -1 \times \mathrm{oi}' - \mathrm{ok}' + \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}')))$

Hiding formulas: 2,

Expanding the definition of shift_bits_left,

Applying bit_split_less

Instantiating the top quantifier in -1 with the terms: $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}')))$, $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n')$, ok', oi',

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_disjoined.1.1.2`.

`cut_bits_overwrite_bits_disjoined.1.2`:

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{oi}' + \mathrm{ok}' \leq \mathrm{ci}'$
{1}	$\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'$ $\mathrm{expt}(2, \mathrm{ci}'))$ $=$ $\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') - \mathrm{rem}(\mathrm{expt}(2, \mathrm{o}$ $\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}') \times \mathrm{expt}(2, \mathrm{ci}' - \mathrm{oi}' - \mathrm{ok}'))$
{2}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') - \mathrm{rem}(\mathrm{expt}(2$ $\mathrm{expt}(2, \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}')))$

Rewriting using expt_plus, matching in *,

This completes the proof of `cut_bits_overwrite_bits_disjoined.1.2`.

`cut_bits_overwrite_bits_disjoined.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') -$ $\mathrm{expt}(2, \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}')))$

Rewriting using rem_ndiv, matching in *,
Rewriting using expt_plus, matching in *,
Applying same_remainder
Instantiating the top quantifier in -1 with the terms: $\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}')$, $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(e$
n',
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`cut_bits_overwrite_bits_disjoined.2.1:`

{-1}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'))$ $(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') - \mathrm{rem}(e$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'))(n')$
{-2}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'),$ $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') -$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$\mathrm{oi}' \geq 0$
{-6}	$\mathrm{ok}' \geq 0$
{-7}	$\mathrm{ci}' \geq 0$
{-8}	$\mathrm{ck}' \geq 0$
{-9}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'))$ $(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))), \mathrm{oi}') +$ $\mathrm{expt}(2, \mathrm{ci}'))$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}')))$

Replacing using formula -1,
Hiding formulas: -1, -2,
Rewriting using ndiv_rem_divisible, matching in *,
we get 3 subgoals:
`cut_bits_overwrite_bits_disjoined.2.1.1:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{rem}(\mathrm{ndiv}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'), \mathrm{expt}(2, \mathrm{ci}')))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}'))) =$ $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{ci}')))$

Rewriting using ndiv_expt_expt, matching in *,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.1.1`.
`cut_bits_overwrite_bits_disjoined.2.1.2`:

```
{-1}  n' >= 0
{-2}  m' >= 0
{-3}  oi' >= 0
{-4}  ok' >= 0
{-5}  ci' >= 0
{-6}  ck' >= 0
{-7}  ci' + ck' <= oi'
------------------------------------------------------------
{1}   divides(expt(2, ci'), expt(2, ci' + ck'))
{2}   ndiv(rem(expt(2, ci' + ck'))(n'), expt(2, ci')) =
       rem(expt(2, ck'))(ndiv(n', expt(2, ci')))
```

Rewriting using expt_divides, matching in *,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.1.2`.
`cut_bits_overwrite_bits_disjoined.2.1.3`:

```
{-1}  n' >= 0
{-2}  m' >= 0
{-3}  oi' >= 0
{-4}  ok' >= 0
{-5}  ci' >= 0
{-6}  ck' >= 0
{-7}  ci' + ck' <= oi'
------------------------------------------------------------
{1}   expt(2, ci') <= expt(2, ci' + ck')
{2}   ndiv(rem(expt(2, ci' + ck'))(n'), expt(2, ci')) =
       rem(expt(2, ck'))(ndiv(n', expt(2, ci')))
```

Using lemma both_sides_expt_gt1_le,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.1.3`.
`cut_bits_overwrite_bits_disjoined.2.2`:

```
{-1}  n' >= 0
{-2}  m' >= 0
{-3}  oi' >= 0
{-4}  ok' >= 0
{-5}  ci' >= 0
{-6}  ck' >= 0
{-7}  ci' + ck' <= oi'
------------------------------------------------------------
{1}   rem(expt(2, ci' + ck'))
        (rem(expt(2, oi'))(n') + shift_bits_left(rem(expt(2, ok'))(ndiv(m', expt(2, oi'))), oi') - rem(expt(2, oi' +
       = rem(expt(2, ci' + ck'))(n')
{2}   divides(expt(2, ci' + ck'),
              rem(expt(2, oi'))(n') + shift_bits_left(rem(expt(2, ok'))(ndiv(m', expt(2, oi'))), oi') - rem(expt(2
{3}   ndiv(rem(expt(2, ci' + ck'))
              (rem(expt(2, oi'))(n') + shift_bits_left(rem(expt(2, ok'))(ndiv(m', expt(2, oi'))), oi') - rem(expt(
           expt(2, ci'))
       = rem(expt(2, ck'))(ndiv(n', expt(2, ci')))
```

Hiding formulas: 1, 3,
Applying divides_sum
Instantiating the top quantifier in -1 with the terms: $\mathrm{rem}(\mathrm{expt}(2,\mathrm{oi}'))(n') - \mathrm{rem}(\mathrm{expt}(2,\mathrm{oi}'+\mathrm{ok}'))(n')$,

shift_bits_left(rem(expt(2, ok′))(ndiv(m′, expt(2, oi′))), oi′), expt(2, ci′ + ck′),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.2.2.1:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\text{oi}' \geq 0$
{-4}	$\text{ok}' \geq 0$
{-5}	$\text{ci}' \geq 0$
{-6}	$\text{ck}' \geq 0$
{-7}	$\text{ci}' + \text{ck}' \leq \text{oi}'$
{1}	divides(expt(2, ci′ + ck′), rem(expt(2, oi′))(n′) − rem(expt(2, oi′ + ok′))(n′))
{2}	divides(expt(2, ci′ + ck′), rem(expt(2, oi′))(n′) + shift_bits_left(rem(expt(2, ok′))(ndiv(m′, expt(2, oi′))), oi′) −

Hiding formulas: 2,

Applying divides_diff

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, \text{oi}'))(n')$, $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$, expt(2, ci′ + ck′),

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: 2,

Splitting conjunctions,

we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.2.2.1.1:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\text{oi}' \geq 0$
{-4}	$\text{ok}' \geq 0$
{-5}	$\text{ci}' \geq 0$
{-6}	$\text{ck}' \geq 0$
{-7}	$\text{ci}' + \text{ck}' \leq \text{oi}'$
{1}	divides(expt(2, ci′ + ck′), $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$)

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: expt(2, oi′ + ok′), n', $\text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying divides_expt_gt

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$, $\text{oi}' + \text{ok}'$, $\text{ci}' + \text{ck}'$,

we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.2.2.1.1.1:`

{-1}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}' \wedge$ $\mathrm{divides}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$ $\supset \mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$
{-2}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$\mathrm{oi}' \geq 0$
{-6}	$\mathrm{ok}' \geq 0$
{-7}	$\mathrm{ci}' \geq 0$
{-8}	$\mathrm{ck}' \geq 0$
{-9}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.2.1.1.1`.
`cut_bits_overwrite_bits_disjoined.2.2.1.1.2:`

{-1}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$\mathrm{oi}' \geq 0$
{-5}	$\mathrm{ok}' \geq 0$
{-6}	$\mathrm{ci}' \geq 0$
{-7}	$\mathrm{ck}' \geq 0$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n') \geq 0$
{2}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$

Hiding formulas: -1, 2,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.2.1.1.2`.
`cut_bits_overwrite_bits_disjoined.2.2.1.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}' + \mathrm{ck}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n'))$

Applying rem_def2
Instantiating the top quantifier in -1 with the terms: $\mathrm{expt}(2, \mathrm{oi}')$, n', $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n')$,
Simplifying, rewriting, and recording with decision procedures,
Applying divides_expt_gt
Instantiating the top quantifier in -1 with the terms: $n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n')$, oi', $\mathrm{ci}' + \mathrm{ck}'$,
we get 2 subgoals:

`cut_bits_overwrite_bits_disjoined.2.2.1.2.1:`

{-1}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' \wedge$ $\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{oi}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n'))$ $\supset\ \mathrm{divides}(\mathrm{expt}(2,\ \mathrm{ci}' + \mathrm{ck}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n'))$
{-2}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{oi}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n'))$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$\mathrm{oi}' \geq 0$
{-6}	$\mathrm{ok}' \geq 0$
{-7}	$\mathrm{ci}' \geq 0$
{-8}	$\mathrm{ck}' \geq 0$
{-9}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{ci}' + \mathrm{ck}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n'))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.2.1.2.1`.
`cut_bits_overwrite_bits_disjoined.2.2.1.2.2:`

{-1}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{oi}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n'))$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$\mathrm{oi}' \geq 0$
{-5}	$\mathrm{ok}' \geq 0$
{-6}	$\mathrm{ci}' \geq 0$
{-7}	$\mathrm{ck}' \geq 0$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') \geq 0$
{2}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{ci}' + \mathrm{ck}'),\ n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n'))$

Hiding formulas: -1, 2,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.2.1.2.2`.
`cut_bits_overwrite_bits_disjoined.2.2.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{ci}' + \mathrm{ck}'),$ $\mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))),\ \mathrm{oi}'))$
{2}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{ci}' + \mathrm{ck}'),$ $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{shift\_bits\_left}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))),\ \mathrm{oi}') -$

Hiding formulas: 2,
Expanding the definition of shift_bits_left,
Rewriting using divides_prod2, matching in *,
Rewriting using expt_divides, matching in *,
This completes the proof of `cut_bits_overwrite_bits_disjoined.2.2.2`.

Q.E.D.

C.17.41 Bits.cut_bits_overwrite_bool_bit_disjoined

Terse proof for `cut_bits_overwrite_bool_bit_disjoined`.

`cut_bits_overwrite_bool_bit_disjoined:`

{1} $\forall$ (n: nat, b: bool, oi, ci, ck: nat):
oi $<$ ci $\vee$ ci + ck $\leq$ oi $\supset$
cut_bits(overwrite_bool_bit(n, b, oi), ci, ck) = cut_bits(n, ci, ck)

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_bool_bit,
Expanding the definition of overwrite_bit,
Rewriting using cut_bits_overwrite_bits_disjoined, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cut_bits_overwrite_bool_bit_disjoined`.
Q.E.D.

C.17.42 Bits.cut_bits_overwrite_bits_contained

Terse proof for `cut_bits_overwrite_bits_contained`.

`cut_bits_overwrite_bits_contained:`

{1} $\forall$ (n, m, oi, ok, ci, ck: nat):
oi $\leq$ ci $\wedge$ ci + ck $\leq$ oi + ok $\supset$
cut_bits(overwrite_bits(n, m, oi, ok), ci, ck) = cut_bits(m, ci, ck)

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_bits,
Expanding the definition of shift_bits_left,
Expanding the definition of cut_bits,
Applying number_split
Instantiating the top quantifier in -1 with the terms: rem(expt(2, ok$'$))(ndiv(m', expt(2, oi$'$))), expt(2, ci$'$ − oi$'$),
we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1:`

{-1} rem(expt(2, ok$'$))(ndiv(m', expt(2, oi$'$))) =
expt(2, ci$'$ − oi$'$) × ndiv(rem(expt(2, ok$'$))(ndiv(m', expt(2, oi$'$))), expt(2, ci$'$ − oi$'$)) + rem(expt(2, ci$'$ −
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} oi$' \geq 0$
{-5} ok$' \geq 0$
{-6} ci$' \geq 0$
{-7} ck$' \geq 0$
{-8} oi$' \leq$ ci$'$
{-9} ci$'$ + ck$' \leq$ oi$'$ + ok$'$

{1} rem(expt(2, ck$'$))
(ndiv(rem(expt(2, oi$'$))(n') + rem(expt(2, ok$'$))(ndiv(m', expt(2, oi$'$))) × expt(2, oi$'$) − rem(expt(2, oi$'$
expt(2, ci$'$)))
= rem(expt(2, ck$'$))(ndiv(m', expt(2, ci$'$)))

Installing automatic rewrites from: expt_divides expt_plus

Rewriting using rem_rem, matching in *,

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1:`

{-1} $\text{rem}(\text{expt}(2,\ \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))) = \text{expt}(2,\ \text{ci}' - \text{oi}') \times \text{ndiv}(\text{rem}(\text{expt}(2,\ \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))),\ \text{expt}(2,\ \text{ci}' - \text{oi}')) + \text{rem}(\text{ex}$

{-2} $n' \geq 0$

{-3} $m' \geq 0$

{-4} $\text{oi}' \geq 0$

{-5} $\text{ok}' \geq 0$

{-6} $\text{ci}' \geq 0$

{-7} $\text{ck}' \geq 0$

{-8} $\text{oi}' \leq \text{ci}'$

{-9} $\text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$

{1} $\text{rem}(\text{expt}(2,\ \text{ck}'))(\text{ndiv}(\text{rem}(\text{expt}(2,\ \text{oi}'))(n') + \text{rem}(\text{expt}(2,\ \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))) \times \text{expt}(2,\ \text{oi}') - \text{rem}($
$\text{expt}(2,\ \text{ci}'))) = \text{rem}(\text{expt}(2,\ \text{ck}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}')))$

Rewriting using ndiv_rem_divisible, matching in -1,

we get 3 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1:`

{-1} $\text{rem}(\text{expt}(2,\ \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))) = \text{expt}(2,\ \text{ci}' - \text{oi}') \times \text{rem}(\text{ndiv}(\text{expt}(2,\ \text{ok}'),\ \text{expt}(2,\ \text{ci}' - \text{oi}')))(\text{ndiv}(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}')),\ \text{exp}$

{-2} $n' \geq 0$

{-3} $m' \geq 0$

{-4} $\text{oi}' \geq 0$

{-5} $\text{ok}' \geq 0$

{-6} $\text{ci}' \geq 0$

{-7} $\text{ck}' \geq 0$

{-8} $\text{oi}' \leq \text{ci}'$

{-9} $\text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$

{1} $\text{rem}(\text{expt}(2,\ \text{ck}'))(\text{ndiv}(\text{rem}(\text{expt}(2,\ \text{oi}'))(n') + \text{rem}(\text{expt}(2,\ \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))) \times \text{expt}(2,\ \text{oi}') - \text{rem}($
$\text{expt}(2,\ \text{ci}'))) = \text{rem}(\text{expt}(2,\ \text{ck}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}')))$

Rewriting using ndiv_times_2, matching in *,

Rewriting using expt_plus, matching in *,

Rewriting using ndiv_expt_expt, matching in *,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using associative_mult, matching in *,

Applying ndiv_plus_mod

Instantiating the top quantifier in -1 with the terms: $n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}')$
$\text{expt}(2,\ \text{ci}'),\ \text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) \times \text{expt}(2, \text{oi}')$,

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1.1:`

{-1} $\text{divides}(\text{expt}(2,\ \text{ci}'),\ n' - \text{rem}(\text{expt}(2,\ \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2,\ \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}'))) \times \text{expt}(2,\ c$
$\supset$
$\text{ndiv}(n' - \text{rem}(\text{expt}(2,\ \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2,\ \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}'))) \times \text{expt}(2,\ \text{ci}') +$
$\text{expt}(2,\ \text{ci}'))$
$=$
$\text{ndiv}(n' - \text{rem}(\text{expt}(2,\ \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2,\ \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}'))) \times \text{expt}(2,\ \text{ci}')$
$\text{expt}(2,\ \text{ci}'))$

{-2} $n' \geq 0$

{-3} $m' \geq 0$

{-4} $\text{oi}' \geq 0$

{-5} $\text{ok}' \geq 0$

{-6} $\text{ci}' \geq 0$

{-7} $\text{ck}' \geq 0$

{-8} $\text{oi}' \leq \text{ci}'$

{-9} $\text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$

{1} $\text{rem}(\text{expt}(2,\ \text{ck}'))$
$(\text{ndiv}(\text{rem}(\text{expt}(2,\ \text{oi}'))(n') + \text{rem}(\text{expt}(2,\ \text{ci}' - \text{oi}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))) \times \text{expt}(2,\ \text{oi}') - \text{rem}(\text{expt}(2,$
$\text{expt}(2,\ \text{ci}')))$
$= \text{rem}(\text{expt}(2,\ \text{ck}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}')))$

Splitting conjunctions,
we get 2 subgoals:
`cut_bits_overwrite_bits_contained.1.1.1.1.1:`

{-1} $\text{ndiv}(n' - \text{rem}(\text{expt}(2,\ \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2,\ \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}'))) \times \text{expt}(2,\ \text{ci}') +$
$\text{expt}(2,\ \text{ci}'))$
$=$
$\text{ndiv}(n' - \text{rem}(\text{expt}(2,\ \text{oi}' + \text{ok}'))(n') + \text{rem}(\text{expt}(2,\ \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}'))) \times \text{expt}(2,\ \text{ci}'),$
$\text{expt}(2,\ \text{ci}'))$

{-2} $n' \geq 0$

{-3} $m' \geq 0$

{-4} $\text{oi}' \geq 0$

{-5} $\text{ok}' \geq 0$

{-6} $\text{ci}' \geq 0$

{-7} $\text{ck}' \geq 0$

{-8} $\text{oi}' \leq \text{ci}'$

{-9} $\text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$

{1} $\text{rem}(\text{expt}(2,\ \text{ck}'))$
$(\text{ndiv}(\text{rem}(\text{expt}(2,\ \text{oi}'))(n') + \text{rem}(\text{expt}(2,\ \text{ci}' - \text{oi}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{oi}'))) \times \text{expt}(2,\ \text{oi}') - \text{rem}(\text{expt}(2,$
$\text{expt}(2,\ \text{ci}')))$
$= \text{rem}(\text{expt}(2,\ \text{ck}'))(\text{ndiv}(m',\ \text{expt}(2,\ \text{ci}')))$

Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Applying ndiv_plus_1
Instantiating the top quantifier in -1 with the terms: $\text{rem}(\text{expt}(2, \text{oi}' - \text{ci}' + \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}'))) \times \text{expt}(2, \text{ci}')$,
$n' - \text{rem}(\text{expt}(2, \text{oi}' + \text{ok}'))(n')$, $\text{expt}(2,\ \text{ci}')$,
Rewriting using divides_prod2, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,

Rewriting using ndiv_times_divident_2, matching in *,
Rewriting using ndiv_self, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on n!1 - rem(expt(2, oi!1 + ok!1))(n!1) = ndiv(n!1, expt(2, oi!1 + ok!1)) * expt(2, oi!1 + ok!1),
we get 2 subgoals:
`cut_bits_overwrite_bits_contained.1.1.1.1.1.1:`

{-1} $n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n') = \mathrm{ndiv}(n',\ \mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')$
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} $\mathrm{oi}' \geq 0$
{-5} $\mathrm{ok}' \geq 0$
{-6} $\mathrm{ci}' \geq 0$
{-7} $\mathrm{ck}' \geq 0$
{-8} $\mathrm{oi}' \leq \mathrm{ci}'$
{-9} $\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$

{1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}'))) + \mathrm{ndiv}(n' - \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}'))(n'),$
$= \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}')))$

Replacing using formula -1,
Hiding formulas: -1,
Rewriting using ndiv_times_divident_2, matching in *,
Rewriting using ndiv_expt_expt, matching in *,
Applying rem_sum_elim1
Instantiating the top quantifier in -1 with the terms: expt(2, ck′), rem(expt(2, oi′ − ci′ + ok′))(ndiv(m′, expt(2, ci′))), ndiv(n′, expt(2, oi′ + ok′)) × expt(2, oi′ − ci′ + ok′), 0,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`cut_bits_overwrite_bits_contained.1.1.1.1.1.1.1:`

{-1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}'))) + \mathrm{ndiv}(n',\ \mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2,$
$= \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}'))))$
{-2} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(n',\ \mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2,\ \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))$
$= \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(0)$
{-3} $n' \geq 0$
{-4} $m' \geq 0$
{-5} $\mathrm{oi}' \geq 0$
{-6} $\mathrm{ok}' \geq 0$
{-7} $\mathrm{ci}' \geq 0$
{-8} $\mathrm{ck}' \geq 0$
{-9} $\mathrm{oi}' \leq \mathrm{ci}'$
{-10} $\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$

{1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}'))) + \mathrm{ndiv}(n',\ \mathrm{expt}(2,\ \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2,$
$= \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}')))$

Replacing using formula -1,
Rewriting using rem_rem, matching in *,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.1.1.1`.

`cut_bits_overwrite_bits_contained.1.1.1.1.1.1.2:`

{-1} $n' \geq 0$
{-2} $m' \geq 0$
{-3} $\mathrm{oi}' \geq 0$
{-4} $\mathrm{ok}' \geq 0$
{-5} $\mathrm{ci}' \geq 0$
{-6} $\mathrm{ck}' \geq 0$
{-7} $\mathrm{oi}' \leq \mathrm{ci}'$
{-8} $\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$

{1} $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$
$(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}'))) + \mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' +$
$=$
$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$
$(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}'))))$

{2} $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$
$(\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))$
$= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(0)$

{3} $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$
$(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}'))) + \mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' +$
$= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}')))$

Hiding formulas: 1, 3,

Rewriting using rem_zero, matching in *,

Rewriting using rem_def2, matching in *,

Rewriting using divides_prod2, matching in *,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.1.1.2`.

`cut_bits_overwrite_bits_contained.1.1.1.1.1.2:`

{-1} $n' \geq 0$
{-2} $m' \geq 0$
{-3} $\mathrm{oi}' \geq 0$
{-4} $\mathrm{ok}' \geq 0$
{-5} $\mathrm{ci}' \geq 0$
{-6} $\mathrm{ck}' \geq 0$
{-7} $\mathrm{oi}' \leq \mathrm{ci}'$
{-8} $\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$

{1} $n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n') =$
$\mathrm{ndiv}(n', \mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')) \times \mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')$

{2} $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$
$(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}'))) + \mathrm{ndiv}(n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'), \mathrm{expt}(2, \mathrm{ci}'$
$= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}')))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.1.2`.

`cut_bits_overwrite_bits_contained.1.1.1.1.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}'),$ $n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n') + \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}'))$
{2}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{rem}(\mathrm{expt}(2, \mathrm{ci}' - \mathrm{oi}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))) \times \mathrm{expt}(2, \mathrm{oi}') -$ $\mathrm{expt}(2, \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}')))$

Hiding formulas: 2,

Rewriting using divides_sum, matching in *,

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1.1.2.1:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}'), n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n'))$
{2}	$\mathrm{divides}(\mathrm{expt}(2, \mathrm{ci}'),$ $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' - \mathrm{ci}' + \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}'))) \times \mathrm{expt}(2, \mathrm{ci}') - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' +$

Hiding formulas: 2,

Applying rem_def2

Instantiating the top quantifier in -1 with the terms: $\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}')$, n', $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n')$,

Simplifying, rewriting, and recording with decision procedures,

Applying divides_expt_gt

Instantiating the top quantifier in -1 with the terms: $n' - \mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}' + \mathrm{ok}'))(n')$, $\mathrm{oi}' + \mathrm{ok}'$, ci',

we get 2 subgoals:

`cut_bits_overwrite_bits_contained.1.1.1.1.2.1.1:`

{-1}	$\mathrm{ci}' \leq \mathrm{oi}' + \mathrm{ok}' \wedge$ divides(expt(2, oi′ + ok′), n′ − rem(expt(2, oi′ + ok′))(n′)) ⊃ divides(expt(2, ci′), n′ − rem(expt(2, oi′ + ok′))(n′))
{-2}	divides(expt(2, oi′ + ok′), n′ − rem(expt(2, oi′ + ok′))(n′))
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$\mathrm{oi}' \geq 0$
{-6}	$\mathrm{ok}' \geq 0$
{-7}	$\mathrm{ci}' \geq 0$
{-8}	$\mathrm{ck}' \geq 0$
{-9}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-10}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	divides(expt(2, ci′), n′ − rem(expt(2, oi′ + ok′))(n′))

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.2.1.1`.
`cut_bits_overwrite_bits_contained.1.1.1.1.2.1.2:`

{-1}	divides(expt(2, oi′ + ok′), n′ − rem(expt(2, oi′ + ok′))(n′))
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$\mathrm{oi}' \geq 0$
{-5}	$\mathrm{ok}' \geq 0$
{-6}	$\mathrm{ci}' \geq 0$
{-7}	$\mathrm{ck}' \geq 0$
{-8}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-9}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	n′ − rem(expt(2, oi′ + ok′))(n′) ≥ 0
{2}	divides(expt(2, ci′), n′ − rem(expt(2, oi′ + ok′))(n′))

Hiding formulas: -1, 2,
Using lemma rem_def_pos,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.2.1.2`.
`cut_bits_overwrite_bits_contained.1.1.1.1.2.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	divides(expt(2, ci′), rem(expt(2, oi′ − ci′ + ok′))(ndiv(m′, expt(2, ci′))) × expt(2, ci′))
{2}	divides(expt(2, ci′), rem(expt(2, oi′ − ci′ + ok′))(ndiv(m′, expt(2, ci′))) × expt(2, ci′) − rem(expt(2, oi′ + ok′))(n′) +

Hiding formulas: 2,
Rewriting using divides_prod2, matching in *,
This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.1.2.2`.

`cut_bits_overwrite_bits_contained.1.1.1.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ci}' - \mathrm{oi}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))) \times \mathrm{expt}(2,\ \mathrm{oi}')$ $< \mathrm{expt}(2,\ \mathrm{ci}')$
{2}	$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ci}' - \mathrm{oi}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))) \times \mathrm{expt}(2,\ \mathrm{oi}') -$ $\mathrm{expt}(2,\ \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}')))$

Hiding formulas: 2,

Applying bit_split_less

Instantiating the top quantifier in -1 with the terms: $\mathrm{rem}(\mathrm{expt}(2, \mathrm{ci}' - \mathrm{oi}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}')))$, $\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n')$, $\mathrm{ci}' - \mathrm{oi}'$, oi',

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.1.2`.

`cut_bits_overwrite_bits_contained.1.1.2:`

{-1}	$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))) =$ $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ci}' - \mathrm{oi}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))) + \mathrm{expt}(2,\ \mathrm{ci}' - \mathrm{oi}') \times \mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$\mathrm{oi}' \geq 0$
{-5}	$\mathrm{ok}' \geq 0$
{-6}	$\mathrm{ci}' \geq 0$
{-7}	$\mathrm{ck}' \geq 0$
{-8}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-9}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	$\mathrm{divides}(\mathrm{expt}(2,\ \mathrm{ci}' - \mathrm{oi}'),\ \mathrm{expt}(2,\ \mathrm{ok}'))$
{2}	$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{oi}'))(n') + \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ok}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{oi}'))) \times \mathrm{expt}(2,\ \mathrm{oi}') - \mathrm{rem}($ $\mathrm{expt}(2,\ \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2,\ \mathrm{ck}'))(\mathrm{ndiv}(m',\ \mathrm{expt}(2,\ \mathrm{ci}')))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.2`.

`cut_bits_overwrite_bits_contained.1.1.3:`

{-1} $\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) =$
$\text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) + \text{expt}(2, \text{ci}' - \text{oi}') \times \text{ndiv}(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}($
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} $\text{oi}' \geq 0$
{-5} $\text{ok}' \geq 0$
{-6} $\text{ci}' \geq 0$
{-7} $\text{ck}' \geq 0$
{-8} $\text{oi}' \leq \text{ci}'$
{-9} $\text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$

{1} $\text{expt}(2, \text{ci}' - \text{oi}') \leq \text{expt}(2, \text{ok}')$
{2} $\text{rem}(\text{expt}(2, \text{ck}'))$
$(\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) \times \text{expt}(2, \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}'$
$\text{expt}(2, \text{ci}')))$
$= \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))$

Hiding formulas: -1, 2,

Using lemma both_sides_expt_gt1_le,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_contained.1.1.3`.

`cut_bits_overwrite_bits_contained.1.2:`

{-1} $\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) =$
$\text{rem}(\text{expt}(2, \text{ci}' - \text{oi}'))(\text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}')))) + \text{expt}(2, \text{ci}' - \text{oi}') \times \text{ndiv}(\text{rem}(\text{expt}(2, c$
{-2} $n' \geq 0$
{-3} $m' \geq 0$
{-4} $\text{oi}' \geq 0$
{-5} $\text{ok}' \geq 0$
{-6} $\text{ci}' \geq 0$
{-7} $\text{ck}' \geq 0$
{-8} $\text{oi}' \leq \text{ci}'$
{-9} $\text{ci}' + \text{ck}' \leq \text{oi}' + \text{ok}'$

{1} $\text{divides}(\text{expt}(2, \text{ci}' - \text{oi}'), \text{expt}(2, \text{ok}'))$
{2} $\text{rem}(\text{expt}(2, \text{ck}'))$
$(\text{ndiv}(\text{rem}(\text{expt}(2, \text{oi}'))(n') + \text{rem}(\text{expt}(2, \text{ok}'))(\text{ndiv}(m', \text{expt}(2, \text{oi}'))) \times \text{expt}(2, \text{oi}') - \text{rem}(\text{expt}(2, \text{oi}'$
$\text{expt}(2, \text{ci}')))$
$= \text{rem}(\text{expt}(2, \text{ck}'))(\text{ndiv}(m', \text{expt}(2, \text{ci}')))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bits_overwrite_bits_contained.1.2`.

`cut_bits_overwrite_bits_contained.2:`

{-1}	$n' \geq 0$
{-2}	$m' \geq 0$
{-3}	$\mathrm{oi}' \geq 0$
{-4}	$\mathrm{ok}' \geq 0$
{-5}	$\mathrm{ci}' \geq 0$
{-6}	$\mathrm{ck}' \geq 0$
{-7}	$\mathrm{oi}' \leq \mathrm{ci}'$
{-8}	$\mathrm{ci}' + \mathrm{ck}' \leq \mathrm{oi}' + \mathrm{ok}'$
{1}	$\mathrm{ci}' - \mathrm{oi}' \geq 0$
{2}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))$ $(\mathrm{ndiv}(\mathrm{rem}(\mathrm{expt}(2, \mathrm{oi}'))(n') + \mathrm{rem}(\mathrm{expt}(2, \mathrm{ok}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{oi}'))) \times \mathrm{expt}(2, \mathrm{oi}') - \mathrm{rem}($ $\mathrm{expt}(2, \mathrm{ci}')))$ $= \mathrm{rem}(\mathrm{expt}(2, \mathrm{ck}'))(\mathrm{ndiv}(m', \mathrm{expt}(2, \mathrm{ci}')))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cut_bits_overwrite_bits_contained.2`.
Q.E.D.

C.17.43 Bits.cut_bits_overwrite_shift_bits_disjoint

Terse proof for `cut_bits_overwrite_shift_bits_disjoint`.

`cut_bits_overwrite_shift_bits_disjoint:`

{1}	$\forall\ (n, m, i_1, k_1, i_2, k_2 \colon \mathrm{nat})\colon$ $i_1 + k_1 \leq i_2 \vee i_2 + k_2 \leq i_1 \supset$ $\mathrm{cut\_bits}(\mathrm{overwrite\_shift\_bits}(n, m, i_1, k_1), i_2, k_2) = \mathrm{cut\_bits}(n, i_2, k_2)$

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_shift_bits,
Rewriting using cut_bits_overwrite_bits_disjoined, matching in *,
This completes the proof of `cut_bits_overwrite_shift_bits_disjoint`.
Q.E.D.

C.17.44 Bits.cut_bits_overwrite_shift_bits_contained_TCC1

Terse proof for `cut_bits_overwrite_shift_bits_contained_TCC1`.

`cut_bits_overwrite_shift_bits_contained_TCC1:`

{1}	$\forall\ (i_1, k_1, i_2, k_2 \colon \mathrm{nat})\colon i_1 \leq i_2 \wedge i_2 + k_2 \leq i_1 + k_1 \supset i_2 - i_1 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cut_bits_overwrite_shift_bits_contained_TCC1`.
Q.E.D.

C.17.45 Bits.cut_bits_overwrite_shift_bits_contained

Terse proof for `cut_bits_overwrite_shift_bits_contained`.

cut_bits_overwrite_shift_bits_contained:

{1} $\forall\ (n,\ m,\ i_1,\ k_1,\ i_2,\ k_2\colon \text{nat}):$
$i_1 \le i_2 \wedge i_2 + k_2 \le i_1 + k_1 \supset$
$\text{cut\_bits}(\text{overwrite\_shift\_bits}(n,\ m,\ i_1,\ k_1),\ i_2,\ k_2) = \text{cut\_bits}(m,\ i_2 - i_1,\ k_2)$

Repeatedly Skolemizing and flattening,
Expanding the definition of overwrite_shift_bits,
Rewriting using cut_bits_overwrite_bits_contained, matching in *,
Rewriting using cut_bits_shift_bits_left, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using zero_cut_bits, matching in *,
This completes the proof of cut_bits_overwrite_shift_bits_contained.
Q.E.D.

C.17.46 Bits.cut_bit_overwrite_shift_bits_TCC1

Terse proof for cut_bit_overwrite_shift_bits_TCC1.

cut_bit_overwrite_shift_bits_TCC1:

{1} $\forall\ (i_1,\ k_1,\ i_2\colon \text{nat})\colon i_1 \le i_2 \wedge i_2 < i_1 + k_1 \supset i_2 - i_1 \ge 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of cut_bit_overwrite_shift_bits_TCC1.
Q.E.D.

C.17.47 Bits.cut_bit_overwrite_shift_bits

Terse proof for cut_bit_overwrite_shift_bits.

cut_bit_overwrite_shift_bits:

{1} $\forall\ (n,\ m,\ i_1,\ k_1,\ i_2\colon \text{nat}):$
$\text{cut\_bit}(\text{overwrite\_shift\_bits}(n,\ m,\ i_1,\ k_1),\ i_2) =$
IF $i_1 \le i_2 \wedge i_2 < i_1 + k_1$
THEN $\text{cut\_bit}(m,\ i_2 - i_1)$
ELSE $\text{cut\_bit}(n,\ i_2)$
ENDIF

Repeatedly Skolemizing and flattening,
Lifting IF-conditions to the top level,
Splitting conjunctions,
we get 2 subgoals:

cut_bit_overwrite_shift_bits.1:

{-1} $n' \ge 0$
{-2} $m' \ge 0$
{-3} $i'_1 \ge 0$
{-4} $k'_1 \ge 0$
{-5} $i'_2 \ge 0$
{1} $i'_1 \le i'_2 \wedge i'_2 < i'_1 + k'_1 \supset$
$\text{cut\_bit}(\text{overwrite\_shift\_bits}(n',\ m',\ i'_1,\ k'_1),\ i'_2) = \text{cut\_bit}(m',\ i'_2 - i'_1)$

Applying disjunctive simplification to flatten sequent,

Rewriting using cut_bit_bits, matching in *,
Rewriting using cut_bits_overwrite_shift_bits_contained, matching in *,
Rewriting using cut_bit_bits, matching in *,
This completes the proof of `cut_bit_overwrite_shift_bits.1`.
`cut_bit_overwrite_shift_bits.2`:

$$
\begin{array}{ll}
\{-1\} & n' \geq 0 \\
\{-2\} & m' \geq 0 \\
\{-3\} & i'_1 \geq 0 \\
\{-4\} & k'_1 \geq 0 \\
\{-5\} & i'_2 \geq 0 \\
\hline
\{1\} & \neg\ (i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1) \supset \\
 & \quad \text{cut\_bit}(\text{overwrite\_shift\_bits}(n',\ m',\ i'_1,\ k'_1),\ i'_2) = \text{cut\_bit}(n',\ i'_2)
\end{array}
$$

Applying disjunctive simplification to flatten sequent,
Rewriting using cut_bit_bits, matching in *,
Rewriting using cut_bits_overwrite_shift_bits_disjoint, matching in *,
we get 2 subgoals:
`cut_bit_overwrite_shift_bits.2.1`:

$$
\begin{array}{ll}
\{-1\} & n' \geq 0 \\
\{-2\} & m' \geq 0 \\
\{-3\} & i'_1 \geq 0 \\
\{-4\} & k'_1 \geq 0 \\
\{-5\} & i'_2 \geq 0 \\
\hline
\{1\} & i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1 \\
\{2\} & \text{IF cut\_bits}(n',\ i'_2,\ 1) = 1 \text{ THEN TRUE ELSE FALSE ENDIF} = \text{cut\_bit}(n',\ i'_2)
\end{array}
$$

Rewriting using cut_bit_bits, matching in *,
This completes the proof of `cut_bit_overwrite_shift_bits.2.1`.
`cut_bit_overwrite_shift_bits.2.2`:

$$
\begin{array}{ll}
\{-1\} & n' \geq 0 \\
\{-2\} & m' \geq 0 \\
\{-3\} & i'_1 \geq 0 \\
\{-4\} & k'_1 \geq 0 \\
\{-5\} & i'_2 \geq 0 \\
\hline
\{1\} & i'_1 + k'_1 \leq i'_2 \vee 1 + i'_2 \leq i'_1 \\
\{2\} & i'_1 \leq i'_2 \wedge i'_2 < i'_1 + k'_1 \\
\{3\} & \text{IF cut\_bits}(\text{overwrite\_shift\_bits}(n',\ m',\ i'_1,\ k'_1),\ i'_2,\ 1) = 1 \\
 & \quad \text{THEN TRUE} \\
 & \text{ELSE FALSE} \\
 & \text{ENDIF} \\
 & \quad = \text{cut\_bit}(n',\ i'_2)
\end{array}
$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cut_bit_overwrite_shift_bits.2.2`.
Q.E.D.

C.17.48 Bits.cut_bit_overwrite_bool_bit

Terse proof for `cut_bit_overwrite_bool_bit`.

`cut_bit_overwrite_bool_bit:`

{1} $\forall$ (n: nat, b: bool, i, j: nat):
cut_bit(overwrite_bool_bit(n, b, i), j) =
IF $i = j$ THEN b ELSE cut_bit(n, j) ENDIF

Repeatedly Skolemizing and flattening,
Lifting IF-conditions to the top level,
Splitting conjunctions,
we get 2 subgoals:
`cut_bit_overwrite_bool_bit.1:`

{-1} $n' \geq 0$
{-2} $i' \geq 0$
{-3} $j' \geq 0$

{1} $i' = j' \supset$ cut_bit(overwrite_bool_bit(n', b', i'), j') = b'

Applying disjunctive simplification to flatten sequent,
Rewriting using cut_bit_bits, matching in *,
Expanding the definition of overwrite_bool_bit,
Expanding the definition of overwrite_bit,
Rewriting using cut_bits_overwrite_bits_contained, matching in *,
Rewriting using cut_bits_shift_bits_left, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on j!1 - i!1 = 0,
we get 2 subgoals:
`cut_bit_overwrite_bool_bit.1.1:`

{-1} $j' - i' = 0$
{-2} $i' = j'$
{-3} $n' \geq 0$
{-4} $i' \geq 0$
{-5} $j' \geq 0$

{1} IF cut_bits(bool_to_nat(b'), $j' - i'$, 1) = 1 THEN TRUE ELSE FALSE ENDIF = b'

Replacing using formula -1,
Hiding formulas: -1,
Expanding the definition of cut_bits,
Expanding the definition of expt,
Expanding the definition of expt,
Rewriting using ndiv_1, matching in *,
Rewriting using rem_mod, matching in *,
Expanding the definition of bool_to_nat,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cut_bit_overwrite_bool_bit.1.1`.
`cut_bit_overwrite_bool_bit.1.2:`

{-1} $i' = j'$
{-2} $n' \geq 0$
{-3} $i' \geq 0$
{-4} $j' \geq 0$

{1} $j' - i' = 0$
{2} IF cut_bits(bool_to_nat(b'), $j' - i'$, 1) = 1 THEN TRUE ELSE FALSE ENDIF = b'

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `cut_bit_overwrite_bool_bit.1.2`.
`cut_bit_overwrite_bool_bit.2:`

$$\begin{array}{ll}\{\text{-1}\} & n' \geq 0\\ \{\text{-2}\} & i' \geq 0\\ \{\text{-3}\} & j' \geq 0\\ \hline \{1\} & \neg\ i' = j' \supset \text{cut\_bit}(\text{overwrite\_bool\_bit}(n',\ b',\ i'),\ j') = \text{cut\_bit}(n',\ j')\end{array}$$

Applying disjunctive simplification to flatten sequent,
Rewriting using cut_bit_bits, matching in *,
Rewriting using cut_bit_bits, matching in *,
Expanding the definition of overwrite_bool_bit,
Expanding the definition of overwrite_bit,
Rewriting using cut_bits_overwrite_bits_disjoined, matching in *,
Hiding formulas: 3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cut_bit_overwrite_bool_bit.2`.
Q.E.D.

C.17.49 Bits.overwrite_shift_bits_merge

Terse proof for `overwrite_shift_bits_merge`.
`overwrite_shift_bits_merge:`

$$\begin{array}{ll}\{1\} & \forall\ (m,\ \text{ci1},\ \text{ck1},\ \text{si},\ \text{ci2},\ \text{ck2},\ \text{os},\ \text{ok}\colon \text{nat})\colon\\ & \quad \text{ci1} + \text{ck1} = \text{ci2} \wedge \text{si} + \text{ck1} = \text{os} \wedge \text{ck2} = \text{ok} \supset\\ & \quad\ \text{overwrite\_shift\_bits}(\text{shift\_bits\_left}(\text{cut\_bits}(m,\ \text{ci1},\ \text{ck1}),\ \text{si}),\ \text{cut\_bits}(m,\ \text{ci2},\ \text{ck2}),\\ & \qquad\qquad\qquad\qquad \text{os},\ \text{ok})\\ & \quad\ = \text{shift\_bits\_left}(\text{cut\_bits}(m,\ \text{ci1},\ \text{ck1} + \text{ck2}),\ \text{si})\end{array}$$

Repeatedly Skolemizing and flattening,
Replacing using formula -11,
Replacing using formula -9,
Replacing using formula -10,
Hiding formulas: -9, -10, -11,
Case splitting on ck2!1 = 0,
we get 2 subgoals:
`overwrite_shift_bits_merge.1:`

$$\begin{array}{ll}\{\text{-1}\} & \text{ck2}' = 0\\ \{\text{-2}\} & m' \geq 0\\ \{\text{-3}\} & \text{ci1}' \geq 0\\ \{\text{-4}\} & \text{ck1}' \geq 0\\ \{\text{-5}\} & \text{si}' \geq 0\\ \{\text{-6}\} & \text{ci1}' + \text{ck1}' \geq 0\\ \{\text{-7}\} & \text{ck2}' \geq 0\\ \{\text{-8}\} & \text{si}' + \text{ck1}' \geq 0\\ \{\text{-9}\} & \text{ck2}' \geq 0\\ \hline \{1\} & \text{overwrite\_shift\_bits}(\text{shift\_bits\_left}(\text{cut\_bits}(m',\ \text{ci1}',\ \text{ck1}'),\ \text{si}'),\\ & \qquad\qquad\qquad\qquad \text{cut\_bits}(m',\ \text{ci1}' + \text{ck1}',\ \text{ck2}'),\ \text{si}' + \text{ck1}',\ \text{ck2}')\\ & \quad\ = \text{shift\_bits\_left}(\text{cut\_bits}(m',\ \text{ci1}',\ \text{ck1}' + \text{ck2}'),\ \text{si}')\end{array}$$

Replacing using formula -1,
Case splitting on cut_bits(m!1, ci1!1 + ck1!1, 0) = 0,

we get 2 subgoals:

`overwrite_shift_bits_merge.1.1:`

{-1}	$\text{cut\_bits}(m', \text{ci1}' + \text{ck1}', 0) = 0$
{-2}	$\text{ck2}' = 0$
{-3}	$m' \geq 0$
{-4}	$\text{ci1}' \geq 0$
{-5}	$\text{ck1}' \geq 0$
{-6}	$\text{si}' \geq 0$
{-7}	$\text{ci1}' + \text{ck1}' \geq 0$
{-8}	$0 \geq 0$
{-9}	$\text{si}' + \text{ck1}' \geq 0$
{-10}	$0 \geq 0$
{1}	$\text{overwrite\_shift\_bits}(\text{shift\_bits\_left}(\text{cut\_bits}(m', \text{ci1}', \text{ck1}'), \text{si}'), \text{cut\_bits}(m', \text{ci1}' + \text{ck1}', 0), \text{si}' + \text{ck1}', 0)$ $= \text{shift\_bits\_left}(\text{cut\_bits}(m', \text{ci1}', \text{ck1}' + 0), \text{si}')$

Replacing using formula -1,

Hiding formulas: -1,

Expanding the definition of overwrite_shift_bits,

Expanding the definition of shift_bits_left,

Rewriting using zero_times1, matching in *,

Expanding the definition of overwrite_bits,

Rewriting using zero_cut_bits, matching in *,

Expanding the definition of shift_bits_left,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.1.1`.

`overwrite_shift_bits_merge.1.2:`

{-1}	$\text{ck2}' = 0$
{-2}	$m' \geq 0$
{-3}	$\text{ci1}' \geq 0$
{-4}	$\text{ck1}' \geq 0$
{-5}	$\text{si}' \geq 0$
{-6}	$\text{ci1}' + \text{ck1}' \geq 0$
{-7}	$0 \geq 0$
{-8}	$\text{si}' + \text{ck1}' \geq 0$
{-9}	$0 \geq 0$
{1}	$\text{cut\_bits}(m', \text{ci1}' + \text{ck1}', 0) = 0$
{2}	$\text{overwrite\_shift\_bits}(\text{shift\_bits\_left}(\text{cut\_bits}(m', \text{ci1}', \text{ck1}'), \text{si}'), \text{cut\_bits}(m', \text{ci1}' + \text{ck1}', 0), \text{si}' + \text{ck1}', 0)$ $= \text{shift\_bits\_left}(\text{cut\_bits}(m', \text{ci1}', \text{ck1}' + 0), \text{si}')$

Hiding formulas: 2,

Rewriting using zero_cut_bits, matching in *,

This completes the proof of `overwrite_shift_bits_merge.1.2`.

`overwrite_shift_bits_merge.2:`

$$
\begin{array}{ll}
\{-1\} & m' \geq 0 \\
\{-2\} & \mathrm{ci1}' \geq 0 \\
\{-3\} & \mathrm{ck1}' \geq 0 \\
\{-4\} & \mathrm{si}' \geq 0 \\
\{-5\} & \mathrm{ci1}' + \mathrm{ck1}' \geq 0 \\
\{-6\} & \mathrm{ck2}' \geq 0 \\
\{-7\} & \mathrm{si}' + \mathrm{ck1}' \geq 0 \\
\{-8\} & \mathrm{ck2}' \geq 0 \\
\hline
\{1\} & \mathrm{ck2}' = 0 \\
\{2\} & \mathrm{overwrite\_shift\_bits}(\mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}'), \mathrm{si}'), \\
 & \qquad \mathrm{cut\_bits}(m', \mathrm{ci1}' + \mathrm{ck1}', \mathrm{ck2}'), \mathrm{si}' + \mathrm{ck1}', \mathrm{ck2}') \\
 & = \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}' + \mathrm{ck2}'), \mathrm{si}')
\end{array}
$$

Expanding the definition of overwrite_shift_bits,
Expanding the definition of overwrite_bits,
Rewriting using cut_bits_shift_bits_left, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using cut_bits_cut_bits, matching in *,
Rewriting using rem_mod, matching in *,
we get 2 subgoals:
`overwrite_shift_bits_merge.2.1:`

$$
\begin{array}{ll}
\{-1\} & m' \geq 0 \\
\{-2\} & \mathrm{ci1}' \geq 0 \\
\{-3\} & \mathrm{ck1}' \geq 0 \\
\{-4\} & \mathrm{si}' \geq 0 \\
\{-5\} & \mathrm{ci1}' + \mathrm{ck1}' \geq 0 \\
\{-6\} & \mathrm{ck2}' \geq 0 \\
\{-7\} & \mathrm{ck1}' + \mathrm{si}' \geq 0 \\
\{-8\} & \mathrm{ck2}' \geq 0 \\
\hline
\{1\} & \mathrm{ck2}' = 0 \\
\{2\} & \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}'), \mathrm{si}') + \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}' + \mathrm{ck1}', \mathrm{ck2}'), \mathrm{ck1}' + \\
 & = \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}' + \mathrm{ck2}'), \mathrm{si}')
\end{array}
$$

Rewriting using rem_mod, matching in *,
we get 2 subgoals:
`overwrite_shift_bits_merge.2.1.1:`

$$
\begin{array}{ll}
\{-1\} & m' \geq 0 \\
\{-2\} & \mathrm{ci1}' \geq 0 \\
\{-3\} & \mathrm{ck1}' \geq 0 \\
\{-4\} & \mathrm{si}' \geq 0 \\
\{-5\} & \mathrm{ci1}' + \mathrm{ck1}' \geq 0 \\
\{-6\} & \mathrm{ck2}' \geq 0 \\
\{-7\} & \mathrm{ck1}' + \mathrm{si}' \geq 0 \\
\{-8\} & \mathrm{ck2}' \geq 0 \\
\hline
\{1\} & \mathrm{ck2}' = 0 \\
\{2\} & \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}'), \mathrm{si}') + \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}' + \mathrm{ck1}', \mathrm{ck2}'), \mathrm{ck1}' + \\
 & = \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}' + \mathrm{ck2}'), \mathrm{si}')
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of shift_bits_left,
Using lemma expt_plus,

Replacing using formula -1,
Hiding formulas: -1,
Rewriting using associative_mult, matching in *,
Applying both_sides_times1
Instantiating the top quantifier in -1 with the terms: expt(2, si′), expt(2, ck1′) × cut_bits(m′, ci1′ + ck1′, ck2′) + cut_bits(m′, c
cut_bits(m′, ci1′, ck1′ + ck2′),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 1, 4,
Expanding the definition of cut_bits,
Installing automatic rewrites from: expt_divides both_sides_times_pos_lt1 ndiv_expt_expt
Applying number_split
Instantiating the top quantifier in -1 with the terms: rem(expt(2, ck1′ + ck2′))(ndiv(m′, expt(2,
ci1′))), expt(2, ck1′),
Replacing using formula -1,
Hiding formulas: -1,
Rewriting using ndiv_rem_divisible, matching in *,
we get 2 subgoals:
`overwrite_shift_bits_merge.2.1.1.1:`

{-1} $m' \geq 0$
{-2} $ci1' \geq 0$
{-3} $ck1' \geq 0$
{-4} $si' \geq 0$
{-5} $ci1' + ck1' \geq 0$
{-6} $ck2' \geq 0$
{-7} $ck1' + si' \geq 0$

{1} $\text{rem}(\text{expt}(2, ck1'))(\text{ndiv}(m', \text{expt}(2, ci1'))) + \text{expt}(2, ck1') \times \text{rem}(\text{expt}(2, ck2'))(\text{ndiv}(m', \text{expt}(2, ci1' + ck$
$=$
$\text{expt}(2, ck1') \times \text{rem}(\text{expt}(2, ck2'))(\text{ndiv}(\text{ndiv}(m', \text{expt}(2, ci1')), \text{expt}(2, ck1'))) + \text{rem}(\text{expt}(2, ck1'))(\text{rem}$
{2} $ck2' = 0$

Rewriting using ndiv_times_2, matching in *,
Rewriting using expt_plus, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using rem_rem, matching in *,
This completes the proof of `overwrite_shift_bits_merge.2.1.1.1`.
`overwrite_shift_bits_merge.2.1.1.2:`

{-1} $m' \geq 0$
{-2} $ci1' \geq 0$
{-3} $ck1' \geq 0$
{-4} $si' \geq 0$
{-5} $ci1' + ck1' \geq 0$
{-6} $ck2' \geq 0$
{-7} $ck1' + si' \geq 0$

{1} $\text{expt}(2, ck1') \leq \text{expt}(2, ck1' + ck2')$
{2} $\text{rem}(\text{expt}(2, ck1'))(\text{ndiv}(m', \text{expt}(2, ci1'))) + \text{expt}(2, ck1') \times \text{rem}(\text{expt}(2, ck2'))(\text{ndiv}(m', \text{expt}(2, ci1' + ck$
$=$
$\text{rem}(\text{expt}(2, ck1'))(\text{rem}(\text{expt}(2, ck1' + ck2'))(\text{ndiv}(m', \text{expt}(2, ci1')))) + \text{expt}(2, ck1') \times \text{ndiv}(\text{rem}(\text{expt}(2,$
{3} $ck2' = 0$

Hiding formulas: 2,
Using lemma both_sides_expt_gt1_le,
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.2.1.1.2`.

`overwrite_shift_bits_merge.2.1.2:`

{-1}	$m' \geq 0$
{-2}	$\mathrm{ci1}' \geq 0$
{-3}	$\mathrm{ck1}' \geq 0$
{-4}	$\mathrm{si}' \geq 0$
{-5}	$\mathrm{ci1}' + \mathrm{ck1}' \geq 0$
{-6}	$\mathrm{ck2}' \geq 0$
{-7}	$\mathrm{ck1}' + \mathrm{si}' \geq 0$
{-8}	$\mathrm{ck2}' \geq 0$
{1}	$\mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}'), \mathrm{si}') < \mathrm{expt}(2, \mathrm{ck1}' + \mathrm{ck2}' + \mathrm{si}')$
{2}	$\mathrm{ck2}' = 0$
{3}	$\mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}' + \mathrm{ck1}', \mathrm{ck2}'), \mathrm{ck1}' + \mathrm{si}') + 2 \times \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck}$ $= \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}' + \mathrm{ck2}'), \mathrm{si}')$

Hiding formulas: 3,

Expanding the definition of shift_bits_left,

Installing automatic rewrites from: expt_plus

Applying both_sides_times_pos_lt1

Instantiating the top quantifier in -1 with the terms: expt(2, si′), cut_bits(m', ci1′, ck1′), expt(2, $\mathrm{ck1}' + \mathrm{ck2}'$),

Simplifying, rewriting, and recording with decision procedures,

Applying both_sides_expt_gt1_le

Instantiating the top quantifier in -1 with the terms: 2, ck1′, $\mathrm{ck1}' + \mathrm{ck2}'$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.2.1.2`.

`overwrite_shift_bits_merge.2.2:`

{-1}	$m' \geq 0$
{-2}	$\mathrm{ci1}' \geq 0$
{-3}	$\mathrm{ck1}' \geq 0$
{-4}	$\mathrm{si}' \geq 0$
{-5}	$\mathrm{ci1}' + \mathrm{ck1}' \geq 0$
{-6}	$\mathrm{ck2}' \geq 0$
{-7}	$\mathrm{ck1}' + \mathrm{si}' \geq 0$
{-8}	$\mathrm{ck2}' \geq 0$
{1}	$\mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}'), \mathrm{si}') < \mathrm{expt}(2, \mathrm{ck1}' + \mathrm{si}')$
{2}	$\mathrm{ck2}' = 0$
{3}	$\mathrm{rem}(\mathrm{expt}(2, \mathrm{ck1}' + \mathrm{si}'))(\mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}'), \mathrm{si}')) + \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m',$ $= \mathrm{shift\_bits\_left}(\mathrm{cut\_bits}(m', \mathrm{ci1}', \mathrm{ck1}' + \mathrm{ck2}'), \mathrm{si}')$

Hiding formulas: 3,

Expanding the definition of shift_bits_left,

Using lemma expt_plus,

Replacing using formula -1,

Hiding formulas: -1,

Using lemma both_sides_times_pos_lt1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `overwrite_shift_bits_merge.2.2`.

Q.E.D.

C.18 Proofs for BitwiseExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.19 Proofs for Block_Disjoint_Rewrites (plain_memory_rewrites.pvs)

C.19.1 Block_Disjoint_Rewrites.different_registers_blocks_disjoint

Terse proof for `different_registers_blocks_disjoint`.

`different_registers_blocks_disjoint:`

```
{1}  ∀ (addr1, addr2: Address, sz1, sz2: nat):
       type_of(addr1) ≠ type_of(addr2) ⊃ blocks_disjoint?(addr1, sz1, addr2, sz2)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `different_registers_blocks_disjoint`.
Q.E.D.

C.20 Proofs for BooleanConversions (conversions.pvs)

C.20.1 BooleanConversions.pointer_to_bool_TCC1

Terse proof for `pointer_to_bool_TCC1`.

`pointer_to_bool_TCC1:`

```
{1}  ∀ (p_typ: Cpp_Subtype(cv(pointer?))): pointer?(cv_base(p_typ))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pointer_to_bool_TCC1`.
Q.E.D.

C.21 Proofs for CatchAux (statements.pvs)

This theory contains no provable formal statements.

C.22 Proofs for Challenge_Device_Memory (device_memory.pvs)

C.22.1 Challenge_Device_Memory.pm_dev_plain

Terse proof for `pm_dev_plain`.

pm_dev_plain:

```
{1}  ∀ (device_read_side_effect,
            device_write_side_effect:
              [[Address, list[Byte], bool] →
                   [Device_memory[Physical_memory, pm_phy, Device_state] →
                        ExprResult[Device_memory, list[Byte]]]],
            pm: Plain_Memory[Device_memory]):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        plain_memory?(pm_phy)
```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of pm_dev_plain.
Q.E.D.

C.22.2 Challenge_Device_Memory.pm_dev_states

Terse proof for pm_dev_states.
pm_dev_states:

```
{1}  ∀ (device_read_side_effect,
            device_write_side_effect:
              [[Address, list[Byte], bool] →
                   [Device_memory[Physical_memory, pm_phy, Device_state] →
                        ExprResult[Device_memory, list[Byte]]]],
            pm: Plain_Memory[Device_memory]):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        pm‘states = em_lift(pm_phy‘states)
```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of pm_dev_states.
Q.E.D.

C.22.3 Challenge_Device_Memory.pm_dev_read

Terse proof for pm_dev_read.
pm_dev_read:

```
{1}  ∀ (device_read_side_effect,
            device_write_side_effect:
              [[Address, list[Byte], bool] →
                   [Device_memory[Physical_memory, pm_phy, Device_state] →
                        ExprResult[Device_memory, list[Byte]]]],
            pm: Plain_Memory[Device_memory]):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        memory_read(pm‘mem) =
         (λ (a: Address):
              em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
```

Expanding the definition of is_device_plain_memory?,
Expanding the definition of device_pm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_dev_read`.
Q.E.D.

C.22.4 Challenge_Device_Memory.pm_dev_write

Terse proof for `pm_dev_write`.

`pm_dev_write`:

```
{1}   ∀ (device_read_side_effect,
           device_write_side_effect:
             [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                   ExprResult[Device_memory, list[Byte]]]],
           pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
         memory_write(pm`mem) =
          (λ (a: Address, b: Byte):
             em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy`mem)(a, b)))
```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
Expanding the definition of device_pm,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_dev_write`.
Q.E.D.

C.22.5 Challenge_Device_Memory.pm_dev_read_side_effect

Terse proof for `pm_dev_read_side_effect`.

`pm_dev_read_side_effect`:

```
{1}   ∀ (device_read_side_effect,
           device_write_side_effect:
             [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                   ExprResult[Device_memory, list[Byte]]]],
           pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
         memory_read_side_effect(pm`mem) =
          (λ (a: Address, bl: list[Byte], cp: bool):
             em_lift[Physical_memory, Device_state, list[Byte]]
               (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
               ## (λ (bl1: list[Byte]): device_read_side_effect(a, bl1, cp)))
```

Expanding the definition of is_device_plain_memory?,

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Expanding the definition of device_pm,
which is trivially true.
This completes the proof of `pm_dev_read_side_effect`.
Q.E.D.

C.22.6 Challenge_Device_Memory.pm_dev_write_side_effect

Terse proof for `pm_dev_write_side_effect`.
`pm_dev_write_side_effect:`

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                    ExprResult[Device_memory, list[Byte]]]],
          pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
         memory_write_side_effect(pm‘mem) =
          (λ (a: Address, bl: list[Byte], cp: bool):
              em_lift[Physical_memory, Device_state, list[Byte]]
                (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
                ## (λ (bl1: list[Byte]): device_write_side_effect(a, bl1, cp)))
```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Expanding the definition of device_pm,
which is trivially true.
This completes the proof of `pm_dev_write_side_effect`.
Q.E.D.

C.22.7 Challenge_Device_Memory.pm_dev_ro_rw_addr

Terse proof for `pm_dev_ro_rw_addr`.
`pm_dev_ro_rw_addr:`

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                    ExprResult[Device_memory, list[Byte]]]],
          pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
         ((pm‘ro_addr ∪ pm‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
```

Expanding the definition of is_device_plain_memory?,
Repeatedly Skolemizing and flattening,
Keeping (-5 -6 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_dev_ro_rw_addr`.
Q.E.D.

C.22.8 Challenge_Device_Memory.device_plain_transformers_ok

Terse proof for `device_plain_transformers_ok`.
`device_plain_transformers_ok`:

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                    ExprResult[Device_memory, list[Byte]]]],
          pm: Plain_Memory[Device_memory]):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        transformers_ok?(pm`states,
                         (memory_read_transformers(pm`mem, (pm`ro_addr ∪ pm`rw_addr)) ∪ memory_write_t
```

Repeatedly Skolemizing and flattening,
Using lemma pm_dev_states,
Using lemma pm_dev_plain,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using transformers_ok_union_transformers, matching in *,
we get 2 subgoals:
`device_plain_transformers_ok.1`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = em_lift(pm_phy`states)
------------------------------------------------------------------------------
{1}   transformers_ok?(pm′`states,
                       memory_read_transformers(pm′`mem, (pm′`ro_addr ∪ pm′`rw_addr)))
{2}   transformers_ok?(pm′`states,
                       (memory_read_transformers(pm′`mem, (pm′`ro_addr ∪ pm′`rw_addr)) ∪ memory_write_tr
```

Hiding formulas: 2,
Using lemma pm_dev_read,
Using lemma em_memory_read_transformers[Physical_memory, Device_state],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Replacing using formula -5,
Rewriting using em_transformers_ok, matching in *,
Using lemma plain_memory_transformers_ok_read_ro_rw,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -5 1) and hiding *,
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_2 gets memory_read_transformers(pm_phy`mem, union(pm_phy`ro_addr, pm_phy`rw_addr)), transformers_1 gets memory_read_transformers(pm_phy`mem, union(pm!1`ro_addr, pm!1`rw_addr)),
Hiding formulas: (-1 2),
Rewriting using memory_read_transformers_mono, matching in *,
Using lemma pm_dev_ro_rw_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_plain_transformers_ok.1`.

`device_plain_transformers_ok.2:`

{-1}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′‘states = em_lift(pm_phy‘states)
{1}	transformers_ok?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))
{2}	transformers_ok?(pm′‘states, (memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ∪ memo

Hiding formulas: 2,

Using lemma pm_dev_write,

Using lemma em_memory_write_transformers[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Replacing using formula -5,

Rewriting using em_transformers_ok, matching in *,

Keeping (-3 -4 1) and hiding *,

Using lemma plain_memory_transformers_ok_write_rw,

Rewriting using transformers_ok_mono_transformers, matching in * where transformers_2 gets memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr), transformers_1 gets memory_write_transformers(pm_phy‘mem, pm!1‘rw_addr),

Rewriting using memory_write_transformers_mono, matching in *,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_transformers_ok.2`.

Q.E.D.

C.22.9 Challenge_Device_Memory.device_plain_unchanged_memory_invariant

Terse proof for `device_plain_unchanged_memory_invariant`.

`device_plain_unchanged_memory_invariant:`

{1}	∀ (device_read_side_effect, device_write_side_effect: [[Address, list[Byte], bool] → [Device_memory[Physical_memory, pm_phy, Device_state] → ExprResult[Device_memory, list[Byte]]]], pm: Plain_Memory[Device_memory]): is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃ unchanged_memory_invariant?(pm‘mem, pm‘states, ((pm‘other_actions ∪ memory_read_transformers(pm‘mem, (p (pm‘ro_addr ∪ pm‘rw_addr))

Repeatedly Skolemizing and flattening,

Rewriting using unchanged_memory_invariant_union_transformers, matching in * where pm gets pm!1‘mem,

we get 2 subgoals:

`device_plain_unchanged_memory_invariant.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
-------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  (pm′`other_actions ∪ memory_read_transformers(pm′`mem,  (pm′`ro_addr ∪
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{2}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  ((pm′`other_actions ∪ memory_read_transformers(pm′`mem,  (pm′`ro_addr ∪
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_memory_read_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Replacing using formula -4,

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

we get 2 subgoals:

`device_plain_unchanged_memory_invariant.1.1:`

```
{-1}  memory_read(pm′`mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-2}  memory_read_transformers(pm′`mem, (pm′`ro_addr ∪ pm′`rw_addr)) =
       em_lift(memory_read_transformers(pm_phy`mem, (pm′`ro_addr ∪ pm′`rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  pm′`states = em_lift(pm_phy`states)
{-5}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′`mem, em_lift(pm_phy`states), pm′`other_actions,
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{2}   unchanged_memory_invariant?(pm′`mem, em_lift(pm_phy`states),
                                  (pm′`other_actions ∪ em_lift(memory_read_transformers(pm_phy`mem,  (pm
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Hiding formulas: 2,

Rewriting using unchanged_memory_invariant_mono, matching in * where transformers_1 gets pm!1`other_actions, transformers_2 gets em_lift[Physical_memory, Device_state](pm_phy`other_actions), addresses_1 gets union(pm!1`ro_addr, pm!1`rw_addr), addresses_2 gets union(pm_phy`ro_addr, pm_phy`rw_addr),

we get 3 subgoals:

`device_plain_unchanged_memory_invariant.1.1.1:`

```
{-1}  memory_read(pm'`mem) =
        (λ (a: Address):
             em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-2}  memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr)) =
       em_lift(memory_read_transformers(pm_phy`mem, (pm'`ro_addr ∪ pm'`rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  pm'`states = em_lift(pm_phy`states)
{-5}  plain_memory?(pm_phy)
------------------------------------------------------------------------------
{1}   (pm'`other_actions ⊆ em_lift[Physical_memory, Device_state](pm_phy`other_actions))
{2}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states), pm'`other_actions,
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Keeping (-3 1) and hiding *,
Expanding the definition of is_device_plain_memory?,
which is trivially true.
This completes the proof of `device_plain_unchanged_memory_invariant.1.1.1`.
`device_plain_unchanged_memory_invariant.1.1.2:`

```
{-1}  memory_read(pm'`mem) =
        (λ (a: Address):
             em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-2}  memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr)) =
       em_lift(memory_read_transformers(pm_phy`mem, (pm'`ro_addr ∪ pm'`rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  pm'`states = em_lift(pm_phy`states)
{-5}  plain_memory?(pm_phy)
------------------------------------------------------------------------------
{1}   ((pm'`ro_addr ∪ pm'`rw_addr) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states), pm'`other_actions,
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Keeping (-3 1) and hiding *,
Using lemma pm_dev_ro_rw_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.1.1.2`.
`device_plain_unchanged_memory_invariant.1.1.3:`

```
{-1}  memory_read(pm'`mem) =
        (λ (a: Address):
             em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-2}  memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr)) =
       em_lift(memory_read_transformers(pm_phy`mem, (pm'`ro_addr ∪ pm'`rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  pm'`states = em_lift(pm_phy`states)
{-5}  plain_memory?(pm_phy)
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states),
                                  em_lift[Physical_memory, De-
      vice_state](pm_phy`other_actions),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states), pm'`other_actions,
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-5 1) and hiding *,
Using lemma plain_memory_unchanged_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma unchanged_memory_invariant_mono[Physical_memory],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`device_plain_unchanged_memory_invariant.1.1.3.1:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------
{1}   ((pm_phy`ro_addr ∪ pm_phy`rw_addr) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, pm_phy`other_actions,
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Rewriting using subset_reflexive, matching in *,
This completes the proof of `device_plain_unchanged_memory_invariant.1.1.3.1`.
`device_plain_unchanged_memory_invariant.1.1.3.2:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------
{1}   (pm_phy`other_actions ⊆ ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_add
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, pm_phy`other_actions,
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.1.1.3.2`.
`device_plain_unchanged_memory_invariant.1.2:`

```
{-1}  memory_read(pm'`mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-2}  memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr)) =
       em_lift(memory_read_transformers(pm_phy`mem, (pm'`ro_addr ∪ pm'`rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  pm'`states = em_lift(pm_phy`states)
{-5}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states),
                                  em_lift(memory_read_transformers(pm_phy`mem,
                                                                   (pm'`ro_addr ∪ pm'`rw_addr))),
                                  (pm'`ro_addr ∪ pm'`rw_addr))
{2}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states),
                                  (pm'`other_actions ∪ em_lift(memory_read_transformers(pm_phy`mem, (pm
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using unchanged_memory_invariant_mono, matching in * where transformers_1 gets memory_read_transformers(pm_phy`mem, union(pm!1`ro_addr, pm!1`rw_addr)), transformers_2 gets memory_read_transformers(pm_phy`mem, union(pm_phy`ro_addr, pm_phy`rw_addr)), addresses_1 gets union(pm!1`ro_addr, pm!1`rw_addr), addresses_2 gets union(pm_phy`ro_addr, pm_phy`rw_addr),

we get 3 subgoals:
`device_plain_unchanged_memory_invariant.1.2.1:`

```
{-1}  memory_read(pm′‘mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-2}  memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) =
        em_lift(memory_read_transformers(pm_phy‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  pm′‘states = em_lift(pm_phy‘states)
{-5}  plain_memory?(pm_phy)
------------------------------------------------------------------------------
{1}   (memory_read_transformers(pm_phy‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ⊆ memory_read_transfor
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_read_transformers(pm_phy‘mem,
                                                           (pm′‘ro_addr ∪ pm′‘rw_addr)),
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{3}   unchanged_memory_invariant?(pm′‘mem, em_lift(pm_phy‘states),
                                  em_lift(memory_read_transformers(pm_phy‘mem,
                                                                   (pm′‘ro_addr ∪ pm′‘rw
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{4}   unchanged_memory_invariant?(pm′‘mem, em_lift(pm_phy‘states),
                                  (pm′‘other_actions ∪ em_lift(memory_read_transformers(pm_phy‘
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Keeping (-3 1) and hiding *,
Rewriting using memory_read_transformers_mono, matching in *,
Using lemma pm_dev_ro_rw_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.1.2.1`.
`device_plain_unchanged_memory_invariant.1.2.2:`

```
{-1}  memory_read(pm′‘mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-2}  memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) =
        em_lift(memory_read_transformers(pm_phy‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  pm′‘states = em_lift(pm_phy‘states)
{-5}  plain_memory?(pm_phy)
------------------------------------------------------------------------------
{1}   ((pm′‘ro_addr ∪ pm′‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_read_transformers(pm_phy‘mem,
                                                           (pm′‘ro_addr ∪ pm′‘rw_addr)),
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{3}   unchanged_memory_invariant?(pm′‘mem, em_lift(pm_phy‘states),
                                  em_lift(memory_read_transformers(pm_phy‘mem,
                                                                   (pm′‘ro_addr ∪ pm′‘rw
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{4}   unchanged_memory_invariant?(pm′‘mem, em_lift(pm_phy‘states),
                                  (pm′‘other_actions ∪ em_lift(memory_read_transformers(pm_phy‘
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Keeping (-3 1) and hiding *,
Using lemma pm_dev_ro_rw_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.1.2.2`.

`device_plain_unchanged_memory_invariant.1.2.3:`

```
{-1}  memory_read(pm'`mem) =
       (λ (a: Address):
            em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-2}  memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr)) =
       em_lift(memory_read_transformers(pm_phy`mem, (pm'`ro_addr ∪ pm'`rw_addr)))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  pm'`states = em_lift(pm_phy`states)
{-5}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_read_transformers(pm_phy`mem,
                                                           (pm_phy`ro_addr ∪ pm_phy`rw_addr)),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_read_transformers(pm_phy`mem,
                                                           (pm'`ro_addr ∪ pm'`rw_addr)),
                                  (pm'`ro_addr ∪ pm'`rw_addr))
{3}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states),
                                  em_lift(memory_read_transformers(pm_phy`mem,
                                                                   (pm'`ro_addr ∪ pm'`rw_addr))),
                                  (pm'`ro_addr ∪ pm'`rw_addr))
{4}   unchanged_memory_invariant?(pm'`mem, em_lift(pm_phy`states),
                                  (pm'`other_actions ∪ em_lift(memory_read_transformers(pm_phy`mem,  (pm
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Keeping (-5 1) and hiding *,

Using lemma plain_memory_unchanged_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_mono[Physical_memory],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_unchanged_memory_invariant.1.2.3.1:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem,  (pm_pl
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------
{1}   ((pm_phy`ro_addr ∪ pm_phy`rw_addr) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_read_transformers(pm_phy`mem,
                                                           (pm_phy`ro_addr ∪ pm_phy`rw_addr)),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Rewriting using subset_reflexive, matching in *,

This completes the proof of `device_plain_unchanged_memory_invariant.1.2.3.1`.

`device_plain_unchanged_memory_invariant.1.2.3.2:`

```
{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘me
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}  plain_memory?(pm_phy)
-------------------------------------------------------------------------------
{1}   (memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)) ⊆ ((pm_phy‘othe
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_read_transformers(pm_phy‘mem,
                                                           (pm_phy‘ro_addr ∪ pm_phy‘rw_ac
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.1.2.3.2`.
`device_plain_unchanged_memory_invariant.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
-------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  (memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Rewriting using unchanged_memory_invariant_union_transformers, matching in * where pm gets pm!1‘mem,
we get 2 subgoals:
`device_plain_unchanged_memory_invariant.2.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
-------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  memory_read_side_effect_super_transformers(pm′‘mem,
                                                                             (pm′‘ro_add
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  (memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{3}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Rewriting using unchanged_memory_invariant_all_transformers, matching in *,
Expanding the definition of memory_read_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Keeping (-2 -4 1) and hiding *,
Using lemma pm_dev_read_side_effect,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Using lemma pm_dev_states,
Using lemma pm_dev_plain,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma fexpr_unchanged_memory_invariant_composition[Device_memory, list[Byte], list[Byte]],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 5 subgoals:
`device_plain_unchanged_memory_invariant.2.1.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = em_lift(pm_phy`states)
{-4}  memory_read_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
            ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl′)):
        unchanged_memory_invariant?(pm′`mem, pm′`states,
                                    singleton(expr_2_super(device_read_side_effect′(a′,
                                                                                    d,
                                                                                    cp′))),
                                    (pm′`ro_addr ∪ pm′`rw_addr))
{2}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(em_lift
                                                           [Physical_memory,
                                                            Device_state,
                                                            list[Byte]]
                                                           (memory_read_side_effect(pm_phy`mem)
                                                              (a′, bl′, cp′))
                                                         ##
                                                         (λ (bl1: list[Byte]):
                                                            device_read_side_effect′(a′,
                                                                                     bl1,
                                                                                     cp′)))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: (-1 2),
Expanding the definition of is_device_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Using lemma unchanged_memory_invariant_mono,
Installing automatic rewrites from: subset_reflexive
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -13 1) and hiding *,
Expanding the definition of drse_super_transformers,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Expanding the definition of member,
Expanding the definition of union,
Expanding the definition of member,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Expanding the definition of singleton,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.1.1`.

`device_plain_unchanged_memory_invariant.2.1.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = em_lift(pm_phy‘states)
{-4}  memory_read_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------
{1}   ∀ (s: (pm′‘states)):
        OK?(em_lift[Physical_memory, Device_state, list[Byte]]
              (memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′))(s))
        ⊃
        data(em_lift[Physical_memory, Device_state, list[Byte]]
               (memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′))(s))
          = bl′
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_read_side_effect(pm
                                                          (a′, bl′, cp′))
                                                        ##
                                                        (λ (bl1: list[Byte]):
                                                           device_read_side_effect′(a′,

                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-6 2),

Using lemma plain_memory_side_effect_content_unchanged_read_block,

Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_unchanged_memory_invariant.2.1.2.1:`

```
{-1}  plain_memory?(pm_phy)
{-2}  side_effect_content_unchanged(address_block(a′, length(bl′)), pm_phy‘states,
                                    memory_read_side_effect(pm_phy‘mem))
{-3}  pm′‘states(s′)
{-4}  OK?(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))
{-5}  OK?(OK(s′
             WITH [(state)
                   := state(memory_read_side_effect(pm_phy‘mem)
                                                   (a′, bl′, cp′)(s′‘state))],
           data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))))
{-6}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-7}  pm′‘states =
       (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy‘states(em‘state))
{-8}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------
{1}   data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state)) = bl′
```

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset_reflexive

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.1.2.1`.

`device_plain_unchanged_memory_invariant.2.1.2.2:`

```
{-1}  plain_memory?(pm_phy)
{-2}  pm′‘states(s′)
{-3}  OK?(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))
{-4}  OK?(OK(s′
             WITH [(state)
                   := state(memory_read_side_effect(pm_phy‘mem)
                                                   (a′, bl′, cp′)(s′‘state))],
           data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))))
{-5}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-6}  pm′‘states =
       (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy‘states(em‘state))
{-7}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------
{1}   (address_block(a′, length(bl′)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}   data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state)) = bl′
```

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets union(pm!1‘ro_addr, pm!1‘rw_addr), c gets union(pm_phy‘ro_addr, pm_phy‘rw_addr),

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.1.2.2`.

`device_plain_unchanged_memory_invariant.2.1.3:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = em_lift(pm_phy‘states)
{-4}  memory_read_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_read_side_effect(pm
                                                          (a′, bl′, cp′)))),
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_read_side_effect(pm
                                                          (a′, bl′, cp′))
                                                       ##
                                                       (λ (bl1: list[Byte]):
                                                          device_read_side_effect′(a′,
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Rewriting using em_lift_expr_2_super, matching in *,

Hiding formulas: 2,

Rewriting using em_lift_singleton, matching in *,

Replacing using formula -3,

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 -6 1) and hiding *,

Using lemma plain_memory_unchanged_read_side_effect_block_ro_rw,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`device_plain_unchanged_memory_invariant.2.1.3.1:`

```
{-1}  plain_memory?(pm_phy)
{-2}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_read_side_effect_super_transformers(pm_phy`mem,
                                                                             address_block
                                                                             (a′, length(bl′))),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_read_side_effect(pm_phy`mem)
                                                         (a′, bl′, cp′))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{3}   (address_block(a′, length(bl′)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets union(pm!1`ro_addr, pm!1`rw_addr), c gets union(pm_phy`ro_addr, pm_phy`rw_addr),

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.1.3.1`.

`device_plain_unchanged_memory_invariant.2.1.3.2:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_read_side_effect_super_transformers(pm_phy`mem,
                                                                             address_block
                                                                             (a′, length(bl′))),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2}  plain_memory?(pm_phy)
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------------------
{1}   ((pm′`ro_addr ∪ pm′`rw_addr) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_read_side_effect(pm_phy`mem)
                                                         (a′, bl′, cp′))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.1.3.2`.

`device_plain_unchanged_memory_invariant.2.1.3.3:`

```
{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_read_side_effect_super_transformers(pm_phy‘mem,
                                                                             address_block
                                                                             (a′, length
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}  plain_memory?(pm_phy)
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------------
{1}   (singleton(expr_2_super(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′))) ⊆ memory_read_sid
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_read_side_effect(pm_phy‘mem)
                                                         (a′, bl′, cp′))),
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.2.1.3.3`.
`device_plain_unchanged_memory_invariant.2.1.4:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = em_lift(pm_phy‘states)
{-4}  memory_read_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states,
                       memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_read_side_effect(pm
                                                          (a′, bl′, cp′))
                                                       ##
                                                       (λ (bl1: list[Byte]):
                                                          device_read_side_effect′(a′,
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Using lemma device_plain_transformers_ok,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.2.1.4`.

`device_plain_unchanged_memory_invariant.2.1.5:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = em_lift(pm_phy`states)
{-4}  memory_read_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]]
                               (memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′))))
        (expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]]
                               (memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′))))
{2}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_read_side_effect(pm_phy`mem)
                                                          (a′, bl′, cp′))
                                                       ##
                                                       (λ (bl1: list[Byte]):
                                                          device_read_side_effect′(a′,
                                                                                   bl1,
                                                                                   cp′)))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Expanding the definition of singleton,
which is trivially true.
This completes the proof of `device_plain_unchanged_memory_invariant.2.1.5`.
`device_plain_unchanged_memory_invariant.2.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  memory_write_side_effect_super_transformers(pm′`mem, pm′`rw_addr),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{2}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  (memory_read_side_effect_super_transformers(pm′`mem, (pm′`ro_addr ∪ pm
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{3}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  ((pm′`other_actions ∪ memory_read_transformers(pm′`mem, (pm′`ro_addr ∪
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Rewriting using unchanged_memory_invariant_all_transformers, matching in *,
Expanding the definition of memory_write_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Keeping (-2 -4 1) and hiding *,
Using lemma pm_dev_write_side_effect,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,

Using lemma pm_dev_states,
Using lemma pm_dev_plain,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma fexpr_unchanged_memory_invariant_composition[Device_memory, list[Byte], list[Byte]],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 5 subgoals:
`device_plain_unchanged_memory_invariant.2.2.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = em_lift(pm_phy‘states)
{-4}  memory_write_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl′)):
        unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                    singleton(expr_2_super(device_write_side_effect′(a′,
                                    (pm′‘ro_addr ∪ pm′‘rw_addr))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  singleton(expr_2_super(em_lift
                                    [Physical_memory,
                                     Device_state,
                                     list[Byte]]
                                    (memory_write_side_effect(pm
                                     (a′, bl′, cp′))
                                  ##
                                  (λ (bl1: list[Byte]):
                                     device_write_side_effect′(a′,
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: (-1 2),
Expanding the definition of is_device_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Using lemma unchanged_memory_invariant_mono,
Installing automatic rewrites from: subset_reflexive
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -13 1) and hiding *,
Expanding the definition of subset?,
Expanding the definition of union,
Expanding the definition of member,
Repeatedly Skolemizing and flattening,
Expanding the definition of dwse_super_transformers,
Instantiating quantified variables,
Expanding the definition of singleton,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.1`.

`device_plain_unchanged_memory_invariant.2.2.2`:

```
{-1}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}  plain_memory?(pm_phy)
{-3}  pm'`states = em_lift(pm_phy`states)
{-4}  memory_write_side_effect(pm'`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
             ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))
{-5}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
{1}   ∀ (s: (pm'`states)):
        OK?(em_lift[Physical_memory, Device_state, list[Byte]]
              (memory_write_side_effect(pm_phy`mem)(a', bl', cp'))(s))
        ⊃
        data(em_lift[Physical_memory, Device_state, list[Byte]]
               (memory_write_side_effect(pm_phy`mem)(a', bl', cp'))(s))
          = bl'
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_write_side_effect(pm_phy`mem
                                                          (a', bl', cp'))
                                                        ##
                                                        (λ (bl1: list[Byte]):
                                                           device_write_side_effect'(a',
                                                                                     bl1,
                                                                                     cp'))))
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-6 2),

Using lemma plain_memory_side_effect_content_unchanged_write_block,

Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_unchanged_memory_invariant.2.2.2.1:`

```
{-1}  plain_memory?(pm_phy)
{-2}  side_effect_content_unchanged(address_block(a′, length(bl′)), pm_phy‘states,
                                    memory_write_side_effect(pm_phy‘mem))
{-3}  pm′‘states(s′)
{-4}  OK?(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))
{-5}  OK?(OK(s′
             WITH [(state)
                   := state(memory_write_side_effect(pm_phy‘mem)
                                                    (a′, bl′, cp′)(s′‘state))],
           data(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))))
{-6}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-7}  pm′‘states =
        (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy‘states(em‘state))
{-8}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
-------------------------------------------------------------------------------
{1}   data(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state)) = bl′
```

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset_reflexive

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.2.1`.

`device_plain_unchanged_memory_invariant.2.2.2.2:`

```
{-1}  plain_memory?(pm_phy)
{-2}  pm′‘states(s′)
{-3}  OK?(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))
{-4}  OK?(OK(s′
             WITH [(state)
                   := state(memory_write_side_effect(pm_phy‘mem)
                                                    (a′, bl′, cp′)(s′‘state))],
           data(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))))
{-5}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-6}  pm′‘states =
        (λ (em: Expand_state[Physical_memory, Device_state]): pm_phy‘states(em‘state))
{-7}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
-------------------------------------------------------------------------------
{1}   (address_block(a′, length(bl′)) ⊆ pm_phy‘rw_addr)
{2}   data(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state)) = bl′
```

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1‘rw_addr, c gets pm_phy‘rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.2.2`.

device_plain_unchanged_memory_invariant.2.2.3:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = em_lift(pm_phy`states)
{-4}  memory_write_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
            ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(em_lift
                                                           [Physical_memory,
                                                             Device_state,
                                                             list[Byte]]
                                                           (memory_write_side_effect(pm_phy`mem
                                                             (a′, bl′, cp′)))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{2}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(em_lift
                                                           [Physical_memory,
                                                             Device_state,
                                                             list[Byte]]
                                                           (memory_write_side_effect(pm_phy`mem
                                                             (a′, bl′, cp′))
                                                         ##
                                                         (λ (bl1: list[Byte]):
                                                            device_write_side_effect′(a′,
                                                                                     bl1,
                                                                                     cp′)))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
```

Hiding formulas: 2,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Replacing using formula -3,

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 -6 1) and hiding *,

Using lemma plain_memory_unchanged_write_side_effect_block_ro_rw,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`device_plain_unchanged_memory_invariant.2.2.3.1:`

{-1}	plain_memory?(pm_phy)
{-2}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-3}	(address_block(a', length(bl′)) ⊆ pm′‘rw_addr)
{1}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, memory_write_side_effect_super_transformers(pm_phy‘mem, address_block(a', length(b (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(expr_2_super(memory_write_side_effect(pm_phy‘mem) (a', bl′, cp′))), (pm′‘ro_addr ∪ pm′‘rw_addr))
{3}	(address_block(a', length(bl′)) ⊆ pm_phy‘rw_addr)

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1‘rw_addr, c gets pm_phy‘rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.3.1`.

`device_plain_unchanged_memory_invariant.2.2.3.2:`

{-1}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, memory_write_side_effect_super_transformers(pm_phy‘mem, address_block(a', length(b (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}	plain_memory?(pm_phy)
{-3}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}	(address_block(a', length(bl′)) ⊆ pm′‘rw_addr)
{1}	((pm′‘ro_addr ∪ pm′‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(expr_2_super(memory_write_side_effect(pm_phy‘mem) (a', bl′, cp′))), (pm′‘ro_addr ∪ pm′‘rw_addr))

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.3.2`.

`device_plain_unchanged_memory_invariant.2.2.3.3:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_write_side_effect_super_transformers(pm_phy`mem,
                                                                              address_block
                                                                              (a',
                                                                               length(bl')),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2}  plain_memory?(pm_phy)
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------------------------
{1}   (singleton(expr_2_super(memory_write_side_effect(pm_phy`mem)(a', bl', cp'))) ⊆ memory_write_side_effect_su
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write_side_effect(pm_phy`mem)
                                                         (a', bl', cp'))),
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant.2.2.3.3`.
`device_plain_unchanged_memory_invariant.2.2.4:`

```
{-1}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-2}  plain_memory?(pm_phy)
{-3}  pm'`states = em_lift(pm_phy`states)
{-4}  memory_write_side_effect(pm'`mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
          em_lift[Physical_memory, Device_state, list[Byte]]
            (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect'(a, bl1, cp)))
{-5}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------------------------
{1}   transformers_ok?(pm'`states,
                       memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr)))
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(em_lift
                                                         [Physical_memory,
                                                          Device_state,
                                                          list[Byte]]
                                                         (memory_write_side_effect(pm_phy`mem
                                                          (a', bl', cp'))
                                                        ##
                                                        (λ (bl1: list[Byte]):
                                                           device_write_side_effect'(a',
                                                                                     bl1,
                                                                                     cp'))))
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Hiding formulas: 2,
Using lemma device_plain_transformers_ok,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_invariant.2.2.4`.
`device_plain_unchanged_memory_invariant.2.2.5`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = em_lift(pm_phy‘states)
{-4}  memory_write_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-5}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------
{1}   singleton(expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]]
                               (memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′))))
              (expr_2_super(em_lift[Physical_memory, Device_state, list[Byte]]
                               (memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′))))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                               singleton(expr_2_super(em_lift
                                                      [Physical_memory,
                                                        Device_state,
                                                        list[Byte]]
                                                      (memory_write_side_effect(pm
                                                        (a′, bl′, cp′))
                                                    ##
                                                    (λ (bl1: list[Byte]):
                                                       device_write_side_effect′(a′,


                               (pm′‘ro_addr ∪ pm′‘rw_addr))
```

Expanding the definition of singleton,
which is trivially true.
This completes the proof of `device_plain_unchanged_memory_invariant.2.2.5`.
Q.E.D.

C.22.10 Challenge_Device_Memory.device_plain_unchanged_memory_invariant_write

Terse proof for `device_plain_unchanged_memory_invariant_write`.
`device_plain_unchanged_memory_invariant_write`:

```
------------------------------------------------------------------------
{1}   ∀ (device_read_side_effect,
           device_write_side_effect:
             [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                   ExprResult[Device_memory, list[Byte]]]],
           pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
          unchanged_memory_invariant?(pm‘mem, pm‘states, mem-
      ory_write_transformers(pm‘mem, pm‘rw_addr),
                                 pm‘ro_addr)
```

Repeatedly Skolemizing and flattening,
Rewriting using em_memory_write_transformers, matching in * where em_pm gets pm!1‘mem, pm gets pm_phy‘mem,
we get 2 subgoals:
`device_plain_unchanged_memory_invariant_write.1:`

{-1} is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)

{1} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
em_lift(memory_write_transformers(pm_phy‘mem, pm′‘rw_addr)),
pm′‘ro_addr)

Using lemma pm_dev_states,
Using lemma pm_dev_plain,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma em_unchanged_memory_invariant,
Using lemma pm_dev_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`device_plain_unchanged_memory_invariant_write.1.1:`

{-1} is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2} memory_read(pm′‘mem) =
(λ (a: Address):
em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-3} plain_memory?(pm_phy)
{-4} pm′‘states = em_lift(pm_phy‘states)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm′‘rw_addr),
(pm_phy‘ro_addr ∪ (pm_phy‘rw_addr \ pm′‘rw_addr)))
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm′‘rw_addr), pm′‘ro_addr)

Hiding formulas: 2,
Rewriting using unchanged_memory_invariant_union_addresses[Physical_memory], matching in *,
we get 2 subgoals:
`device_plain_unchanged_memory_invariant_write.1.1.1:`

{-1} is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2} memory_read(pm′‘mem) =
(λ (a: Address):
em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-3} plain_memory?(pm_phy)
{-4} pm′‘states = em_lift(pm_phy‘states)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm′‘rw_addr), pm_phy‘ro_ad
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm′‘rw_addr),
(pm_phy‘ro_addr ∪ (pm_phy‘rw_addr \ pm′‘rw_addr)))

Using lemma unchanged_memory_invariant_mono,
Rewriting using subset_reflexive, matching in *,
Using lemma plain_memory_unchanged_memory_invariant_write,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using memory_write_transformers_mono, matching in *,
Expanding the definition of is_device_plain_memory?,
which is trivially true.
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.1.1`.
`device_plain_unchanged_memory_invariant_write.1.1.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read(pm′‘mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-3}  plain_memory?(pm_phy)
{-4}  pm′‘states = em_lift(pm_phy‘states)
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm′‘rw_addr),
                                  (pm_phy‘rw_addr \ pm′‘rw_addr))
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm′‘rw_addr),
                                  (pm_phy‘ro_addr ∪ (pm_phy‘rw_addr \ pm′‘rw_addr)))
```

Rewriting using unchanged_memory_invariant_all_transformers, matching in *,
Expanding the definition of memory_write_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Hiding formulas: (-3 2 3),
Using lemma plain_memory_unchanged_memory_write_invariant,
Expanding the definition of unchanged_memory_write_invariant?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`device_plain_unchanged_memory_invariant_write.1.1.2.1:`

```
{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(a′, b′))),
                                  (pm_phy‘rw_addr \ {a′}))
{-2}  b′ < max_byte
{-3}  pm′‘rw_addr(a′)
{-4}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-5}  memory_read(pm′‘mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-6}  plain_memory?(pm_phy)
{-7}  pm′‘states = em_lift(pm_phy‘states)
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(a′, b′))),
                                  (pm_phy‘rw_addr \ pm′‘rw_addr))
```

Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`device_plain_unchanged_memory_invariant_write.1.1.2.1.1:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(a′, b′))),
                                  (pm_phy`rw_addr \ {a′}))
{-2}  b′ < max_byte
{-3}  pm′`rw_addr(a′)
{-4}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-5}  memory_read(pm′`mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-6}  plain_memory?(pm_phy)
{-7}  pm′`states = em_lift(pm_phy`states)
------------------------------------------------------------------------------
{1}   ((pm_phy`rw_addr \ pm′`rw_addr) ⊆ (pm_phy`rw_addr \ {a′}))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(a′, b′))),
                                  (pm_phy`rw_addr \ pm′`rw_addr))
```

Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.1.2.1.1`.
`device_plain_unchanged_memory_invariant_write.1.1.2.1.2:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(a′, b′))),
                                  (pm_phy`rw_addr \ {a′}))
{-2}  b′ < max_byte
{-3}  pm′`rw_addr(a′)
{-4}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-5}  memory_read(pm′`mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-6}  plain_memory?(pm_phy)
{-7}  pm′`states = em_lift(pm_phy`states)
------------------------------------------------------------------------------
{1}   (singleton(expr_2_super(memory_write(pm_phy`mem)(a′, b′))) ⊆ singleton(expr_2_super(memory_write(pm_ph
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(a′, b′))),
                                  (pm_phy`rw_addr \ pm′`rw_addr))
```

Rewriting using subset_reflexive, matching in *,
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.1.2.1.2`.
`device_plain_unchanged_memory_invariant_write.1.1.2.2:`

```
{-1}  b′ < max_byte
{-2}  pm′`rw_addr(a′)
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  memory_read(pm′`mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = em_lift(pm_phy`states)
------------------------------------------------------------------------------
{1}   pm_phy`rw_addr(a′)
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(a′, b′))),
                                  (pm_phy`rw_addr \ pm′`rw_addr))
```

Expanding the definition of is_device_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-2 -5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.1.2.2`.
`device_plain_unchanged_memory_invariant_write.1.2`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read(pm′‘mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-3}  plain_memory?(pm_phy)
{-4}  pm′‘states = em_lift(pm_phy‘states)
------------------------------------------------------------------------------
{1}   (pm′‘ro_addr ⊆ (pm_phy‘ro_addr ∪ (pm_phy‘rw_addr \ pm′‘rw_addr)))
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm′‘rw_addr), pm
```

Expanding the definition of is_device_plain_memory?,
which is trivially true.
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.2`.
`device_plain_unchanged_memory_invariant_write.1.3`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read(pm′‘mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-3}  plain_memory?(pm_phy)
{-4}  pm′‘states = em_lift(pm_phy‘states)
------------------------------------------------------------------------------
{1}   (memory_write_transformers(pm_phy‘mem, pm′‘rw_addr) ⊆ memory_write_transformers(pm_phy‘m
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm′‘rw_addr), pm
```

Rewriting using memory_write_transformers_mono, matching in *,
Rewriting using subset_reflexive, matching in *,
This completes the proof of `device_plain_unchanged_memory_invariant_write.1.3`.
`device_plain_unchanged_memory_invariant_write.2`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
------------------------------------------------------------------------------
{1}   (pm′‘mem‘memory_write =
         (λ (a: Address, b: Byte):
            em_lift[Physical_memory, Device_state, Unit](pm_phy‘mem‘memory_write(a, b))))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  memory_write_transformers(pm′‘mem, pm′‘rw_addr), pm′‘ro
```

Using lemma pm_dev_write,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_plain_unchanged_memory_invariant_write.2`.
Q.E.D.

C.22.11 Challenge_Device_Memory.device_plain_unchanged_memory_write_invariant

Terse proof for `device_plain_unchanged_memory_write_invariant`.

`device_plain_unchanged_memory_write_invariant:`

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                    ExprResult[Device_memory, list[Byte]]]],
          pm: Plain_Memory[Device_memory]):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        unchanged_memory_write_invariant?(pm`mem, pm`states, pm`rw_addr)
```

Expanding the definition of unchanged_memory_write_invariant?,

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_write,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma em_unchanged_memory_invariant[Physical_memory, Device_state],

Using lemma pm_dev_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_memory_write_invariant,

Expanding the definition of unchanged_memory_write_invariant?,

Instantiating quantified variables,

Using lemma unchanged_memory_invariant_mono,

Rewriting using subset_reflexive, matching in *,

Case splitting on subset?(pm!1`rw_addr, pm_phy`rw_addr),

we get 2 subgoals:

`device_plain_unchanged_memory_write_invariant.1:`

$$
\begin{array}{|ll}
\{\text{-}1\} & (\mathrm{pm}'\text{`rw\_addr} \subseteq \mathrm{pm\_phy}\text{`rw\_addr}) \\
\{\text{-}2\} & ((\mathrm{pm}'\text{`rw\_addr} \setminus \{\mathrm{waddr}'\}) \subseteq (\mathrm{pm\_phy}\text{`rw\_addr} \setminus \{\mathrm{waddr}'\})) \wedge \\
 & \quad \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm\_phy}\text{`mem},\ \mathrm{pm\_phy}\text{`states}, \\
 & \qquad \mathrm{singleton}(\mathrm{expr\_2\_super}(\mathrm{memory\_write}(\mathrm{pm\_phy}\text{`mem}) \\
 & \qquad\qquad (\mathrm{waddr}',\ b'))), \\
 & \qquad (\mathrm{pm\_phy}\text{`rw\_addr} \setminus \{\mathrm{waddr}'\})) \\
 & \quad \supset \\
 & \quad \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm\_phy}\text{`mem},\ \mathrm{pm\_phy}\text{`states}, \\
 & \qquad \mathrm{singleton}(\mathrm{expr\_2\_super}(\mathrm{memory\_write}(\mathrm{pm\_phy}\text{`mem}) \\
 & \qquad\qquad (\mathrm{waddr}',\ b'))), \\
 & \qquad (\mathrm{pm}'\text{`rw\_addr} \setminus \{\mathrm{waddr}'\})) \\
\{\text{-}3\} & \mathrm{pm\_phy}\text{`rw\_addr}(\mathrm{waddr}') \supset \\
 & \quad \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm\_phy}\text{`mem},\ \mathrm{pm\_phy}\text{`states}, \\
 & \qquad \mathrm{singleton}(\mathrm{expr\_2\_super}(\mathrm{memory\_write}(\mathrm{pm\_phy}\text{`mem}) \\
 & \qquad\qquad (\mathrm{waddr}',\ b'))), \\
 & \qquad (\mathrm{pm\_phy}\text{`rw\_addr} \setminus \{\mathrm{waddr}'\})) \\
\{\text{-}4\} & \mathrm{is\_device\_plain\_memory?}(\mathrm{device\_read\_side\_effect}',\ \mathrm{device\_write\_side\_effect}')(\mathrm{pm}') \\
\{\text{-}5\} & \mathrm{memory\_read}(\mathrm{pm}'\text{`mem}) = \\
 & \quad (\lambda\ (a\colon \mathrm{Address})\colon \\
 & \qquad \mathrm{em\_lift}\big[\mathrm{Physical\_memory},\ \mathrm{Device\_state},\ \mathrm{Byte}\big](\mathrm{memory\_read}(\mathrm{pm\_phy}\text{`mem})(a))) \\
\{\text{-}6\} & \mathrm{plain\_memory?}(\mathrm{pm\_phy}) \\
\{\text{-}7\} & \mathrm{pm}'\text{`states} = \mathrm{em\_lift}(\mathrm{pm\_phy}\text{`states}) \\
\{\text{-}8\} & \mathrm{memory\_write}(\mathrm{pm}'\text{`mem}) = \\
 & \quad (\lambda\ (a\colon \mathrm{Address},\ b\colon \mathrm{Byte})\colon \\
 & \qquad \mathrm{em\_lift}\big[\mathrm{Physical\_memory},\ \mathrm{Device\_state},\ \mathrm{Unit}\big](\mathrm{memory\_write}(\mathrm{pm\_phy}\text{`mem})(a,\ b))) \\
\{\text{-}9\} & b' < \mathrm{max\_byte} \\
\{\text{-}10\} & \mathrm{pm}'\text{`rw\_addr}(\mathrm{waddr}') \\
\hline
\{1\} & \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm\_phy}\text{`mem},\ \mathrm{pm\_phy}\text{`states}, \\
 & \qquad \mathrm{singleton}(\mathrm{expr\_2\_super}(\mathrm{memory\_write}(\mathrm{pm\_phy}\text{`mem}) \\
 & \qquad\qquad (\mathrm{waddr}',\ b'))), \\
 & \qquad (\mathrm{pm}'\text{`rw\_addr} \setminus \{\mathrm{waddr}'\})) \\
\{2\} & \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm}'\text{`mem},\ \mathrm{pm}'\text{`states}, \\
 & \qquad \mathrm{em\_lift}(\mathrm{singleton}(\mathrm{expr\_2\_super}(\mathrm{memory\_write}(\mathrm{pm\_phy}\text{`mem}) \\
 & \qquad\qquad (\mathrm{waddr}',\ b')))), \\
 & \qquad (\mathrm{pm}'\text{`rw\_addr} \setminus \{\mathrm{waddr}'\}))
\end{array}
$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_unchanged_memory_write_invariant.1.1:`

```
{-1}  (pm′`rw_addr ⊆ pm_phy`rw_addr)
{-2}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-3}  memory_read(pm′`mem) =
        (λ (a: Address):
           em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy`mem)(a)))
{-4}  plain_memory?(pm_phy)
{-5}  pm′`states = em_lift(pm_phy`states)
{-6}  memory_write(pm′`mem) =
        (λ (a: Address, b: Byte):
           em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy`mem)(a, b)))
{-7}  b′ < max_byte
{-8}  pm′`rw_addr(waddr′)
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)
                                                         (waddr′, b′))),
                                  (pm_phy`rw_addr \ {waddr′}))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)
                                                         (waddr′, b′))),
                                  (pm′`rw_addr \ {waddr′}))
{3}   pm_phy`rw_addr(waddr′)
{4}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  em_lift(singleton(expr_2_super(memory_write(pm_phy`mem)
                                                                 (waddr′, b′)))),
                                  (pm′`rw_addr \ {waddr′}))
```

Keeping (-1 -8 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_write_invariant.1.1`.

`device_plain_unchanged_memory_write_invariant.1.2:`

```
{-1}  (pm′‘rw_addr ⊆ pm_phy‘rw_addr)
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)
                                                         (waddr′, b′))),
                                  (pm_phy‘rw_addr \ {waddr′}))
{-3}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}  memory_read(pm′‘mem) =
       (λ (a: Address):
            em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))
{-5}  plain_memory?(pm_phy)
{-6}  pm′‘states = em_lift(pm_phy‘states)
{-7}  memory_write(pm′‘mem) =
       (λ (a: Address, b: Byte):
            em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy‘mem)(a, b)))
{-8}  b′ < max_byte
{-9}  pm′‘rw_addr(waddr′)
------------------------------------------------------------------------------
{1}   ((pm′‘rw_addr \ {waddr′}) ⊆ (pm_phy‘rw_addr \ {waddr′}))
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)
                                                         (waddr′, b′))),
                                  (pm′‘rw_addr \ {waddr′}))
{3}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  em_lift(singleton(expr_2_super(memory_write(pm_phy‘mem)
                                                                 (waddr′, b′)))),
                                  (pm′‘rw_addr \ {waddr′}))
```

Expanding the definition of subset?,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of remove,

Expanding the definition of /=,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of member,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_write_invariant.1.2`.

`device_plain_unchanged_memory_write_invariant.2:`

{-1} ((pm′‘rw_addr \ {waddr′}) ⊆ (pm_phy‘rw_addr \ {waddr′})) ∧
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)
(waddr′, b′))),
(pm_phy‘rw_addr \ {waddr′}))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)
(waddr′, b′))),
(pm′‘rw_addr \ {waddr′}))

{-2} pm_phy‘rw_addr(waddr′) ⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)
(waddr′, b′))),
(pm_phy‘rw_addr \ {waddr′}))

{-3} is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)

{-4} memory_read(pm′‘mem) =
(λ (a: Address):
em_lift[Physical_memory, Device_state, Byte](memory_read(pm_phy‘mem)(a)))

{-5} plain_memory?(pm_phy)

{-6} pm′‘states = em_lift(pm_phy‘states)

{-7} memory_write(pm′‘mem) =
(λ (a: Address, b: Byte):
em_lift[Physical_memory, Device_state, Unit](memory_write(pm_phy‘mem)(a, b)))

{-8} b′ < max_byte

{-9} pm′‘rw_addr(waddr′)

{1} (pm′‘rw_addr ⊆ pm_phy‘rw_addr)

{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)
(waddr′, b′))),
(pm′‘rw_addr \ {waddr′}))

{3} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
em_lift(singleton(expr_2_super(memory_write(pm_phy‘mem)
(waddr′, b′)))),
(pm′‘rw_addr \ {waddr′}))

Expanding the definition of is_device_plain_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_unchanged_memory_write_invariant.2`.

Q.E.D.

C.22.12 Challenge_Device_Memory.device_plain_changed_memory_invariant

Terse proof for `device_plain_changed_memory_invariant`.

`device_plain_changed_memory_invariant:`

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                 [Device_memory[Physical_memory, pm_phy, Device_state] →
                      ExprResult[Device_memory, list[Byte]]]],
          pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
         changed_memory_invariant?(pm‘mem, pm‘states, pm‘rw_addr)
```

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Using lemma pm_dev_read,

Using lemma pm_dev_write,

Using lemma em_changed_memory_invariant[Physical_memory, Device_state],

Using lemma plain_memory_changed_memory_invariant,

Using lemma changed_memory_invariant_mono,

Expanding the definition of is_device_plain_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_changed_memory_invariant`.

Q.E.D.

C.22.13 Challenge_Device_Memory.device_plain_read_side_effect_unchanged

Terse proof for `device_plain_read_side_effect_unchanged`.

`device_plain_read_side_effect_unchanged:`

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                 [Device_memory[Physical_memory, pm_phy, Device_state] →
                      ExprResult[Device_memory, list[Byte]]]],
          pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
         side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
                                       memory_read_side_effect(pm‘mem))
```

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_read_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using side_effect_content_unchanged_composition, matching in * where states gets pm!1‘states,

we get 3 subgoals:

device_plain_read_side_effect_unchanged.1:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
            ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
-------------------------------------------------------------------------------
{1}   transformer_invariant?(pm′`states,
                             se_super_transformers((pm′`ro_addr ∪ pm′`rw_addr),
                                                   λ (a: Address, bl: list[Byte], cp: bool):
                                                     em_lift[Physical_memory, Device_state, list[Byte]]
                                                       (memory_read_side_effect(pm_phy`mem)(a, bl, cp))))
{2}   side_effect_content_unchanged((pm′`ro_addr ∪ pm′`rw_addr), pm′`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, Device_state, list[Byte]]
                                        (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
                                      ##
                                      (λ (bl1: list[Byte]):
                                         device_read_side_effect′(a, bl1, cp)))
```

Hiding formulas: 2,
Using lemma pm_dev_plain,
Using lemma pm_dev_states,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_invariant,
Using lemma transformer_invariant_all_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of se_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Rewriting using em_lift_expr_2_super, matching in *,
Rewriting using em_lift_singleton, matching in *,
Using lemma em_transformer_invariant[Physical_memory, Device_state],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -5 -6 1) and hiding *,
Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(memory_read_side_effect(pm_phy`mem) (a!1, bl!1, cp!1))), transformers_2 gets union(pm_phy`other_actions, union(union(memory_read_transformers(pm_phy`mem, union (pm_phy`ro_addr, pm_phy`rw_addr)), memory_write_transformers(pm_phy`mem, pm_phy`rw_addr)), union(memory_read_side_effect_super_transformers(pm_phy`mem, union (pm_phy`ro_addr, pm_phy`rw_addr)), memory_write_side_effect_super_transformers(pm_phy`mem, pm_phy`rw_addr)))),
Using lemma pm_dev_ro_rw_addr,
Using lemma subset_transitive,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Expanding the definition of member,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of singleton,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_read_side_effect_unchanged.1`.

`device_plain_read_side_effect_unchanged.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
------------------------------------------------------------------------------
{1}   side_effect_content_unchanged((pm′‘ro_addr ∪ pm′‘rw_addr), pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                        (memory_read_side_effect(pm_phy‘mem)(a, bl, cp)))
{2}   side_effect_content_unchanged((pm′‘ro_addr ∪ pm′‘rw_addr), pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                        (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
                                      ##
                                      (λ (bl1: list[Byte]):
                                         device_read_side_effect′(a, bl1, cp)))
```

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma side_effect_content_unchanged_mono[Device_memory],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Rewriting using em_side_effect_content_unchanged, matching in *,

Expanding the definition of plain_memory?,

which is trivially true.

This completes the proof of `device_plain_read_side_effect_unchanged.2`.

`device_plain_read_side_effect_unchanged.3`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
            ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
-------------------------------------------------------------------------------
{1}   side_effect_content_unchanged((pm′`ro_addr ∪ pm′`rw_addr), pm′`states,
                                    device_read_side_effect′)
{2}   side_effect_content_unchanged((pm′`ro_addr ∪ pm′`rw_addr), pm′`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                          (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
                                       ##
                                       (λ (bl1: list[Byte]):
                                          device_read_side_effect′(a, bl1, cp)))
```

Hiding formulas: 2,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_read_side_effect_unchanged.3`.

Q.E.D.

C.22.14 Challenge_Device_Memory.device_plain_write_side_effect_unchanged

Terse proof for `device_plain_write_side_effect_unchanged`.

`device_plain_write_side_effect_unchanged`:

```
-------------------------------------------------------------------------------
{1}   ∀ (device_read_side_effect,
           device_write_side_effect:
             [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                   ExprResult[Device_memory, list[Byte]]]],
           pm: Plain_Memory[Device_memory]):
        is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
          side_effect_content_unchanged(pm`rw_addr, pm`states, mem-
      ory_write_side_effect(pm`mem))
```

Repeatedly Skolemizing and flattening,

Using lemma pm_dev_write_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using side_effect_content_unchanged_composition, matching in * where states gets pm!1`states,

we get 3 subgoals:

```
device_plain_write_side_effect_unchanged.1:

{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm′‘states,
                             se_super_transformers(pm′‘rw_addr,
                                                   λ (a: Address, bl: list[Byte], cp: bool):
                                                     em_lift[Physical_memory, Device_state, list[Byte]]
                                                       (memory_write_side_effect(pm_phy‘
                                                          (a, bl, cp))))
{2}   side_effect_content_unchanged(pm′‘rw_addr, pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, Device_state, list[Byte]]
                                        (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
                                      ##
                                      (λ (bl1: list[Byte]):
                                         device_write_side_effect′(a, bl1, cp)))
```

Hiding formulas: 2,

Using lemma pm_dev_plain,

Using lemma pm_dev_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_all_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma em_transformer_invariant[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 -6 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(memory_write_side_effect(pm_phy‘mem) (a!1, bl!1, cp!1))), transformers_2 gets union(pm_phy‘other_actions, union(union(memory_read_transformers(pm_phy‘mem, union (pm_phy‘ro_addr, pm_phy‘rw_addr)), memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr)), union(memory_read_side_effect_super_transformers(pm_phy‘mem, union (pm_phy‘ro_addr, pm_phy‘rw_addr)), memory_write_side_effect_super_transformers(pm_phy‘mem, pm_phy‘rw_addr)))),

Using lemma subset_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_write_side_effect_unchanged.1.1:`

```
{-1}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-2}  (address_block(a', length(bl')) ⊆ pm_phy`rw_addr)
{-3}  transformer_invariant?(pm_phy`states,
                             (pm_phy`other_actions ∪ ((memory_read_transformers(pm_phy`mem, (pm_phy`ro_
{-4}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
------------------------------------------------------------------------------------------------------
{1}   (singleton(expr_2_super(memory_write_side_effect(pm_phy`mem)(a', bl', cp'))) ⊆ (pm_phy`other_actions ∪ ((m
{2}   transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(memory_write_side_effect(pm_phy`mem)
                                                    (a', bl', cp'))))
```

Keeping (-2 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_write_side_effect_super_transformers,

Expanding the definition of singleton,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_write_side_effect_unchanged.1.1`.

`device_plain_write_side_effect_unchanged.1.2:`

```
{-1}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-2}  transformer_invariant?(pm_phy`states,
                             (pm_phy`other_actions ∪ ((memory_read_transformers(pm_phy`mem, (pm_phy`ro_
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
------------------------------------------------------------------------------------------------------
{1}   (pm'`rw_addr ⊆ pm_phy`rw_addr)
{2}   (singleton(expr_2_super(memory_write_side_effect(pm_phy`mem)(a', bl', cp'))) ⊆ (pm_phy`other_actions ∪ ((m
{3}   transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(memory_write_side_effect(pm_phy`mem)
                                                    (a', bl', cp'))))
```

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_write_side_effect_unchanged.1.2`.

`device_plain_write_side_effect_unchanged.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′‘mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
            em_lift[Physical_memory, Device_state, list[Byte]]
                 (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
             ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
------------------------------------------------------------------------
{1}   side_effect_content_unchanged(pm′‘rw_addr, pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                           (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
{2}   side_effect_content_unchanged(pm′‘rw_addr, pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                           (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
                                     ##
                                     (λ (bl1: list[Byte]):
                                          device_write_side_effect′(a, bl1, cp)))
```

Hiding formulas: 2,
Using lemma pm_dev_plain,
Using lemma pm_dev_states,
Using lemma pm_dev_ro_rw_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma side_effect_content_unchanged_mono[Device_memory],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`device_plain_write_side_effect_unchanged.2.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  ((pm′‘ro_addr ∪ pm′‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-3}  pm′‘states = em_lift(pm_phy‘states)
{-4}  plain_memory?(pm_phy)
{-5}  memory_write_side_effect(pm′‘mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
            em_lift[Physical_memory, Device_state, list[Byte]]
                 (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
             ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
------------------------------------------------------------------------
{1}   side_effect_content_unchanged(pm_phy‘rw_addr, pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                           (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
{2}   side_effect_content_unchanged(pm′‘rw_addr, pm′‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                           (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
```

Replacing using formula -3,
Rewriting using em_side_effect_content_unchanged, matching in *,

Expanding the definition of plain_memory?,
which is trivially true.
This completes the proof of `device_plain_write_side_effect_unchanged.2.1`.
`device_plain_write_side_effect_unchanged.2.2`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  ((pm′`ro_addr ∪ pm′`rw_addr) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-3}  pm′`states = em_lift(pm_phy`states)
{-4}  plain_memory?(pm_phy)
{-5}  memory_write_side_effect(pm′`mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
            em_lift[Physical_memory, Device_state, list[Byte]]
                (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
             ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
-------------------------------------------------------------------------------
{1}   (pm′`rw_addr ⊆ pm_phy`rw_addr)
{2}   side_effect_content_unchanged(pm′`rw_addr, pm′`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                          (memory_write_side_effect(pm_phy`mem)(a, bl, cp)))
```

Expanding the definition of is_device_plain_memory?,
which is trivially true.
This completes the proof of `device_plain_write_side_effect_unchanged.2.2`.
`device_plain_write_side_effect_unchanged.3`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′`mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
            em_lift[Physical_memory, Device_state, list[Byte]]
                (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
             ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
-------------------------------------------------------------------------------
{1}   side_effect_content_unchanged(pm′`rw_addr, pm′`states, device_write_side_effect′)
{2}   side_effect_content_unchanged(pm′`rw_addr, pm′`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                          (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
                                       ##
                                       (λ (bl1: list[Byte]):
                                            device_write_side_effect′(a, bl1, cp)))
```

Hiding formulas: 2,
Expanding the definition of is_device_plain_memory?,
which is trivially true.
This completes the proof of `device_plain_write_side_effect_unchanged.3`.
Q.E.D.

C.22.15 Challenge_Device_Memory.device_plain_transformers_ok2

Terse proof for `device_plain_transformers_ok2`.

device_plain_transformers_ok2:

```
{1}  ∀ (device_read_side_effect,
        device_write_side_effect:
          [[Address, list[Byte], bool] →
             [Device_memory[Physical_memory, pm_phy, Device_state] →
                ExprResult[Device_memory, list[Byte]]]],
        pm: Plain_Memory[Device_memory]):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        transformers_ok?(pm‘states,
                         (memory_read_side_effect_super_transformers(pm‘mem, (pm‘ro_addr ∪ pm
```

Repeatedly Skolemizing and flattening,

Rewriting using transformers_ok_all_transformers, matching in *,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

device_plain_transformers_ok2.1:

```
{-1}  memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr))(q′)
{-2}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
------------------------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states, singleton(q′))
{2}   transformers_ok?(pm′‘states,
                       (memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘
```

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 -4 1) and hiding *,

Using lemma pm_dev_read_side_effect,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using fexpr_composition_transformers_ok, matching in * where states gets pm!1‘states, P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`device_plain_transformers_ok2.1.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_read_side_effect(pm_phy`mem)
                                                 (a′, bl′, cp′)))))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_read_side_effect(pm_phy`mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_read_side_effect′(a′, bl1, cp′)))))
```

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma em_transformers_ok[Physical_memory, Device_state],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_transformers_ok_read_side_effects_ro_rw,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -3 -6 1) and hiding *,

Rewriting using transformers_ok_mono_transformers[Physical_memory], matching in * where transformers_1 gets singleton(expr_2_super(memory_read_side_effect(pm_phy`mem) (a!1, bl!1, cp!1))), transformers_2 gets memory_read_side_effect_super_transformers(pm_phy`mem, union (pm_phy`ro_addr, pm_phy`rw_addr)),

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -4 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of singleton,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets union(pm_phy`ro_addr, pm_phy`rw_addr), b gets union(pm!1`ro_addr, pm!1`rw_addr),

This completes the proof of `device_plain_transformers_ok2.1.1`.

`device_plain_transformers_ok2.1.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
-------------------------------------------------------------------------------
{1}   transformer_invariant?(pm′‘states,
                             singleton(expr_2_super(em_lift
                                                    [Physical_memory, De-
      vice_state, list[Byte]]
                                                    (memory_read_side_effect(pm_phy‘
                                                      (a′, bl′, cp′)))))
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_read_side_effect(pm_phy‘mem)
                                                (a′, bl′, cp′))
                                             ##
                                             (λ (bl1: list[Byte]):
                                                device_read_side_effect′(a′, bl1, cp′)))))
```

Hiding formulas: 2,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Rewriting using em_transformer_invariant, matching in *,

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -7 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of singleton,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets union(pm_phy‘ro_addr, pm_phy‘rw_addr), b gets union(pm!1‘ro_addr, pm!1‘rw_addr),

This completes the proof of `device_plain_transformers_ok2.1.2`.

`device_plain_transformers_ok2.1.3:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′‘mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
          em_lift[Physical_memory, Device_state, list[Byte]]
            (memory_read_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
-------------------------------------------------------------------------------
{1}   ∀ (s: (pm′‘states)):
        OK?(em_lift[Physical_memory, Device_state, list[Byte]]
              (memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′))(s))
         ⊃
         data(em_lift[Physical_memory, Device_state, list[Byte]]
                (memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′))(s))
          = bl′
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_read_side_effect(pm_phy‘mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_read_side_effect′(a′, bl1, cp′)))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -5 2),

Using lemma pm_dev_plain,

Using lemma pm_dev_ro_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_transformers_ok2.1.3.1:`

{-1}	side_effect_content_unchanged((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), pm_phy‘states, memory_read_side_effect(pm_phy‘mem))
{-2}	OK?(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))
{-3}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}	((pm′‘ro_addr ∪ pm′‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-5}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr) pm_phy‘ro_addr)
{-7}	unchanged_memory_write_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-8}	changed_memory_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-9}	transformers_ok?(pm_phy‘states, ((memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_
{-10}	side_effect_content_unchanged(pm_phy‘rw_addr, pm_phy‘states, memory_write_side_effect(pm_phy‘mem))
{-11}	pm′‘states(s′)
{-12}	(address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{1}	(address_block(a′, length(bl′)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}	data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state)) = bl′

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets union(pm_phy‘ro_addr, pm_phy‘rw_addr), b gets union(pm!1‘ro_addr, pm!1‘rw_addr),

This completes the proof of `device_plain_transformers_ok2.1.3.1`.

`device_plain_transformers_ok2.1.3.2:`

{-1}	side_effect_content_unchanged((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), pm_phy‘states, memory_read_side_effect(pm_phy‘mem))
{-2}	OK?(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state))
{-3}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-4}	((pm′‘ro_addr ∪ pm′‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-5}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6}	unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr) pm_phy‘ro_addr)
{-7}	unchanged_memory_write_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-8}	changed_memory_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-9}	transformers_ok?(pm_phy‘states, ((memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_
{-10}	side_effect_content_unchanged(pm_phy‘rw_addr, pm_phy‘states, memory_write_side_effect(pm_phy‘mem))
{-11}	pm′‘states(s′)
{-12}	(address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{1}	pm_phy‘states(s′‘state)
{2}	data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′‘state)) = bl′

Using lemma pm_dev_states,

Expanding the definition of em_lift,

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_transformers_ok2.1.3.2`.
`device_plain_transformers_ok2.1.4`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_read_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_read_side_effect(pm_phy`mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_read_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl′)):
        transformers_ok?(pm′`states,
                         singleton(expr_2_super(device_read_side_effect′(a′, d, cp′))))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_read_side_effect(pm_phy`mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_read_side_effect′(a′, bl1, cp′)))))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of is_device_plain_memory?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-14 1) and hiding *,
Expanding the definition of subset?,
Expanding the definition of member,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Expanding the definition of drse_super_transformers,
Expanding the definition of union,
Expanding the definition of member,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_plain_transformers_ok2.1.4`.
`device_plain_transformers_ok2.2`:

```
{-1}  memory_write_side_effect_super_transformers(pm′`mem, pm′`rw_addr)(q′)
{-2}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
------------------------------------------------------------------------
{1}   transformers_ok?(pm′`states, singleton(q′))
{2}   transformers_ok?(pm′`states,
                       (memory_read_side_effect_super_transformers(pm′`mem,  (pm′`ro_addr ∪ pm′`rw_addr)) ∪
```

Hiding formulas: 2,
Expanding the definition of memory_write_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Keeping (-2 -4 1) and hiding *,
Using lemma pm_dev_write_side_effect,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using fexpr_composition_transformers_ok, matching in * where states gets pm!1‘states, P gets LAMBDA (bl: list[Byte]): bl = bl!1,
we get 4 subgoals:
`device_plain_transformers_ok2.2.1:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_write_side_effect(pm_phy‘mem)
                                                 (a′, bl′, cp′)))))
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_write_side_effect(pm_phy‘mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_write_side_effect′(a′, bl1, cp′)))))
```

Rewriting using em_lift_expr_2_super, matching in *,
Rewriting using em_lift_singleton, matching in *,
Using lemma pm_dev_states,
Using lemma pm_dev_plain,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma em_transformers_ok[Physical_memory, Device_state],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_transformers_ok_write_side_effects_ro_rw,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -3 -6 1) and hiding *,
Rewriting using transformers_ok_mono_transformers[Physical_memory], matching in * where transformers_1 gets singleton(expr_2_super(memory_write_side_effect(pm_phy‘mem) (a!1, bl!1, cp!1))), transformers_2 gets memory_write_side_effect_super_transformers(pm_phy‘mem, pm_phy‘rw_addr),
Keeping (-2 -3 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Expanding the definition of member,
Expanding the definition of singleton,
Expanding the definition of memory_write_side_effect_super_transformers,
Instantiating quantified variables,
Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets pm_phy‘rw_addr, b gets pm!1‘rw_addr,
Expanding the definition of is_device_plain_memory?,
which is trivially true.
This completes the proof of `device_plain_transformers_ok2.2.1`.

`device_plain_transformers_ok2.2.2:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′`mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy`mem)(a, bl, cp))
            ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
------------------------------------------------------------------------------
{1}   transformer_invariant?(pm′`states,
                             singleton(expr_2_super(em_lift
                                                    [Physical_memory, De-
      vice_state, list[Byte]]
                                                    (memory_write_side_effect(pm_phy`mem)
                                                       (a′, bl′, cp′)))))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_write_side_effect(pm_phy`mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_write_side_effect′(a′, bl1, cp′)))))
```

Hiding formulas: 2,

Rewriting using em_lift_expr_2_super, matching in *,

Rewriting using em_lift_singleton, matching in *,

Using lemma pm_dev_states,

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -3,

Rewriting using em_transformer_invariant, matching in *,

Using lemma plain_memory_invariant,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 -6 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of singleton,

Expanding the definition of union,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets pm_phy`rw_addr, b gets pm!1`rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_transformers_ok2.2.2`.

`device_plain_transformers_ok2.2.3:`

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′‘mem) =
       (λ (a: Address, bl: list[Byte], cp: bool):
          em_lift[Physical_memory, Device_state, list[Byte]]
            (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------
{1}   ∀ (s: (pm′‘states)):
        OK?(em_lift[Physical_memory, Device_state, list[Byte]]
              (memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′))(s))
        ⊃
        data(em_lift[Physical_memory, Device_state, list[Byte]]
               (memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′))(s))
          = bl′
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
vice_state, list[Byte]]
                                              (memory_write_side_effect(pm_phy‘mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_write_side_effect′(a′, bl1, cp′)))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -5 2),

Using lemma pm_dev_plain,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Using lemma side_effect_content_unchanged_content,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`device_plain_transformers_ok2.2.3.1:`

```
{-1}  side_effect_content_unchanged(pm_phy`rw_addr, pm_phy`states,
                                    memory_write_side_effect(pm_phy`mem))
{-2}  OK?(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'`state))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_write_transformers(pm_phy`mem, pm_phy`rw_addr),
                                  pm_phy`ro_addr)
{-6}  unchanged_memory_write_invariant?(pm_phy`mem, pm_phy`states, pm_phy`rw_addr)
{-7}  changed_memory_invariant?(pm_phy`mem, pm_phy`states, pm_phy`rw_addr)
{-8}  transformers_ok?(pm_phy`states,
                       ((memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)) ∪ me
{-9}  side_effect_content_unchanged((pm_phy`ro_addr ∪ pm_phy`rw_addr), pm_phy`states,
                                    memory_read_side_effect(pm_phy`mem))
{-10} pm'`states(s')
{-11} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
 {1}  (address_block(a', length(bl')) ⊆ pm_phy`rw_addr)
 {2}  data(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'`state)) = bl'
```

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), c gets pm_phy`rw_addr, b gets pm!1`rw_addr,

Expanding the definition of is_device_plain_memory?,

which is trivially true.

This completes the proof of `device_plain_transformers_ok2.2.3.1`.

`device_plain_transformers_ok2.2.3.2:`

```
{-1}  side_effect_content_unchanged(pm_phy`rw_addr, pm_phy`states,
                                    memory_write_side_effect(pm_phy`mem))
{-2}  OK?(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'`state))
{-3}  is_device_plain_memory?(device_read_side_effect', device_write_side_effect')(pm')
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_write_transformers(pm_phy`mem, pm_phy`rw_addr),
                                  pm_phy`ro_addr)
{-6}  unchanged_memory_write_invariant?(pm_phy`mem, pm_phy`states, pm_phy`rw_addr)
{-7}  changed_memory_invariant?(pm_phy`mem, pm_phy`states, pm_phy`rw_addr)
{-8}  transformers_ok?(pm_phy`states,
                       ((memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)) ∪ me
{-9}  side_effect_content_unchanged((pm_phy`ro_addr ∪ pm_phy`rw_addr), pm_phy`states,
                                    memory_read_side_effect(pm_phy`mem))
{-10} pm'`states(s')
{-11} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
 {1}  pm_phy`states(s'`state)
 {2}  data(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'`state)) = bl'
```

Using lemma pm_dev_states,

Expanding the definition of em_lift,

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_transformers_ok2.2.3.2`.

`device_plain_transformers_ok2.2.4`:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{-2}  memory_write_side_effect(pm′‘mem) =
        (λ (a: Address, bl: list[Byte], cp: bool):
           em_lift[Physical_memory, Device_state, list[Byte]]
             (memory_write_side_effect(pm_phy‘mem)(a, bl, cp))
           ## (λ (bl1: list[Byte]): device_write_side_effect′(a, bl1, cp)))
{-3}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl′)):
        transformers_ok?(pm′‘states,
                         singleton(expr_2_super(device_write_side_effect′(a′, d, cp′))))
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(em_lift[Physical_memory, De-
      vice_state, list[Byte]]
                                              (memory_write_side_effect(pm_phy‘mem)
                                                 (a′, bl′, cp′))
                                            ##
                                            (λ (bl1: list[Byte]):
                                               device_write_side_effect′(a′, bl1, cp′)))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of is_device_plain_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-14 1) and hiding *,

Expanding the definition of subset?,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Expanding the definition of dwse_super_transformers,

Expanding the definition of union,

Expanding the definition of member,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `device_plain_transformers_ok2.2.4`.

Q.E.D.

C.22.16 Challenge_Device_Memory.device_memory_plain_memory

Terse proof for `device_memory_plain_memory`.

device_memory_plain_memory:

```
{1}  ∀ (device_read_side_effect,
          device_write_side_effect:
            [[Address, list[Byte], bool] →
                [Device_memory[Physical_memory, pm_phy, Device_state] →
                    ExprResult[Device_memory, list[Byte]]]],
          pm):
       is_device_plain_memory?(device_read_side_effect, device_write_side_effect)(pm) ⊃
        plain_memory?(pm)
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 7 subgoals:
device_memory_plain_memory.1:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}   side_effect_content_unchanged(pm′‘rw_addr, pm′‘states, mem-
      ory_write_side_effect(pm′‘mem))
```

Using lemma device_plain_write_side_effect_unchanged,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of device_memory_plain_memory.1.
device_memory_plain_memory.2:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}   side_effect_content_unchanged((pm′‘ro_addr ∪ pm′‘rw_addr), pm′‘states,
                                                  memory_read_side_effect(pm′‘mem))
```

Using lemma device_plain_read_side_effect_unchanged,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of device_memory_plain_memory.2.
device_memory_plain_memory.3:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}   transformers_ok?(pm′‘states,
                       ((memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ∪ memory_write_t
```

Using lemma device_plain_transformers_ok,
Using lemma device_plain_transformers_ok2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using transformers_ok_union_transformers, matching in * where states gets pm!1‘states, transformers_1 gets union(memory_read_transformers(pm!1‘mem, union(pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr)), transformers_2 gets union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr)),
This completes the proof of device_memory_plain_memory.3.
device_memory_plain_memory.4:

```
{-1}  is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}   changed_memory_invariant?(pm′‘mem, pm′‘states, pm′‘rw_addr)
```

Using lemma device_plain_changed_memory_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of device_memory_plain_memory.4.

`device_memory_plain_memory.5`:

{-1}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}	unchanged_memory_write_invariant?(pm′‘mem, pm′‘states, pm′‘rw_addr)

Using lemma device_plain_unchanged_memory_write_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_memory_plain_memory.5`.
`device_memory_plain_memory.6`:

{-1}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}	unchanged_memory_invariant?(pm′‘mem, pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr), pm′‘ro

Using lemma device_plain_unchanged_memory_invariant_write,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_memory_plain_memory.6`.
`device_memory_plain_memory.7`:

{-1}	is_device_plain_memory?(device_read_side_effect′, device_write_side_effect′)(pm′)
{1}	unchanged_memory_invariant?(pm′‘mem, pm′‘states, ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm (pm′‘ro_addr ∪ pm′‘rw_addr))

Using lemma device_plain_unchanged_memory_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `device_memory_plain_memory.7`.
Q.E.D.

C.23 Proofs for Challenge_Random_Device (random_device.pvs)

C.23.1 Challenge_Random_Device.IMP_Random_Device_TCC1

Terse proof for `IMP_Random_Device_TCC1`.
`IMP_Random_Device_TCC1`:

{1}	$0 < \text{size}(\text{uidt}(\text{dt\_uint})) \wedge 3 \times \text{size}(\text{uidt}(\text{dt\_uint})) \leq \text{max\_linear\_offset}$

Using lemma dt_uint_size,
This completes the proof of `IMP_Random_Device_TCC1`.
Q.E.D.

C.23.2 Challenge_Random_Device.in_blessed_memory_disjoint_device

Terse proof for `in_blessed_memory_disjoint_device`.
`in_blessed_memory_disjoint_device`:

{1}	$\forall$ (a: Address, l: nat): (address_block(a, l) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃ disjoint?(address_block(a, l), address_block(base, size))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `in_blessed_memory_disjoint_device`.
Q.E.D.

C.23.3 Challenge_Random_Device.in_blessed_memory_not_device_address

Terse proof for `in_blessed_memory_not_device_address`.
`in_blessed_memory_not_device_address:`

{1} $\forall$ (a: Address, bl: list[Byte]):
(address_block(a, length(bl)) $\subseteq$ (pm`ro_addr $\cup$ pm`rw_addr)) $\wedge$ cons?(bl) $\supset$
$a \neq$ rnd_seed $\wedge$ $a \neq$ mem_cnt $\wedge$ $a \neq$ rnd_val

Repeatedly Skolemizing and flattening,
Using lemma in_blessed_memory_disjoint_device,
Replacing using formula -3,
Keeping (-1 -4 1) and hiding *,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
Adding type constraints for base,
Expanding the definition of disjoint?,
Expanding the definition of empty?,
Instantiating the top quantifier in -5 with the terms: (a!1),
Case splitting on size(uidt(dt_uint)) > 0,
we get 2 subgoals:
`in_blessed_memory_not_device_address.1:`

{-1} size(uidt(dt_uint)) > 0
{-2} Mem?(base`type_of)
{-3} 0 $\leq$ base`offset
{-4} base`offset < max_linear_offset
{-5} base + size[Physical_memory, pm_phy] $\leq$ max_linear
{-6} cons?(bl$'$)

{1} ($a' \in$ (address_block(a', 1 + length(cdr(bl$'$))) $\cap$ address_block(base, size)))
{2} $a' \neq$ rnd_seed $\wedge$ $a' \neq$ mem_cnt $\wedge$ $a' \neq$ rnd_val

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `in_blessed_memory_not_device_address.1`.
`in_blessed_memory_not_device_address.2:`

{-1} Mem?(base`type_of)
{-2} 0 $\leq$ base`offset
{-3} base`offset < max_linear_offset
{-4} base + size[Physical_memory, pm_phy] $\leq$ max_linear
{-5} cons?(bl$'$)

{1} size(uidt(dt_uint)) > 0
{2} ($a' \in$ (address_block(a', 1 + length(cdr(bl$'$))) $\cap$ address_block(base, size)))
{3} $a' \neq$ rnd_seed $\wedge$ $a' \neq$ mem_cnt $\wedge$ $a' \neq$ rnd_val

Using lemma dt_uchar_size,
Using lemma dt_uint_size,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `in_blessed_memory_not_device_address.2`.
Q.E.D.

C.23.4 Challenge_Random_Device.random_side_effects_unchanged_invariant

Terse proof for `random_side_effects_unchanged_invariant`.
`random_side_effects_unchanged_invariant:`

```
|
------------------------------------------------------------------------------
{1}  unchanged_memory_invariant?(pm‘mem, pm‘states,
                                 (drse_super_transformers((pm‘ro_addr ∪ pm‘rw_addr), rnd_dev_r
                                 (pm‘ro_addr ∪ pm‘rw_addr))
```

Installing automatic rewrites from: pm random_device_pm random_device_pm_mem
Rewriting using unchanged_memory_invariant_all_transformers, matching in *,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using em_const_unchanged_memory_invariant, matching in * where em_pm gets device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect), pm gets pm_phy‘mem,
we get 2 subgoals:
`random_side_effects_unchanged_invariant.1:`

```
{-1}  union(drse_super_transformers(((pm_phy‘ro_addr \ address_block(base, size)) ∪ (pm_phy‘rw_addr \
                                    rnd_dev_read_side_effect),
            dwse_super_transformers((pm_phy‘rw_addr \ address_block(base, size)),
                                    rnd_dev_write_side_effect))
      (q′)
------------------------------------------------------------------------------
{1}  (device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect)‘memory_read =
       (λ (a: Address):
          em_lift[Physical_memory, Random_device_state[Physical_memory, pm_phy], Byte]
            (pm_phy‘mem‘memory_read(a))))
{2}  unchanged_memory_invariant?(device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect),
                                 em_lift(pm_phy‘states), singleton(q′),
                                 ((pm_phy‘ro_addr \ address_block(base, size)) ∪ (pm_phy‘rw_ad
```

Expanding the definition of device_pm,
which is trivially true.
This completes the proof of `random_side_effects_unchanged_invariant.1`.
`random_side_effects_unchanged_invariant.2:`

```
{-1}  union(drse_super_transformers(((pm_phy‘ro_addr \ address_block(base, size)) ∪ (pm_phy‘rw_addr \
                                    rnd_dev_read_side_effect),
            dwse_super_transformers((pm_phy‘rw_addr \ address_block(base, size)),
                                    rnd_dev_write_side_effect))
      (q′)
------------------------------------------------------------------------------
{1}  ∀ (s: Expand_state[Physical_memory, Random_device_state[Physical_memory, pm_phy]]):
       has_next_state(q′(s)) ⊃ state(q′(s))‘state = s‘state
{2}  unchanged_memory_invariant?(device_pm(rnd_dev_read_side_effect, rnd_dev_write_side_effect),
                                 em_lift(pm_phy‘states), singleton(q′),
                                 ((pm_phy‘ro_addr \ address_block(base, size)) ∪ (pm_phy‘rw_ad
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of union,

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`random_side_effects_unchanged_invariant.2.1:`

{-1} has_next_state($q'(s')$)
{-2} drse_super_transformers(((pm_phy‘ro_addr \ address_block(base, size)) ∪ (pm_phy‘rw_addr \ address_block(bas
rnd_dev_read_side_effect)
(q')

{1} state($q'(s')$)‘state = s'‘state

Expanding the definition of drse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 1) and hiding *,

Rewriting using has_next_state_expr_2_super, matching in *,

Rewriting using state_expr_2_super, matching in *,

Expanding the definition of rnd_dev_read_side_effect,

Expanding the definition of unaligned_access,

Case splitting on null?(bl!1) OR disjoint?(address_block(a!1, length(bl!1)), address_block(base, size[Physical_memory, pm_phy])) OR (NOT cp!1 AND length(bl!1) = size(uidt(dt_uint)) AND (a!1 = rnd_seed OR a!1 = mem_cnt OR a!1 = rnd_val)),

we get 2 subgoals:

`random_side_effects_unchanged_invariant.2.1.1:`

```
{-1}  null?(bl′) ∨
       disjoint?(address_block(a′, length(bl′)),
                 address_block(base, size[Physical_memory, pm_phy]))
        ∨
        (¬ cp′ ∧
           length(bl′) = size(uidt(dt_uint)) ∧
            (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
{-2}  has_next_state((IF null?(bl′) ∨
                              disjoint?(address_block(a′, length(bl′)),
                                        address_block(base, size))
                               ∨
                               (¬ cp′ ∧
                                  length(bl′) = size(uidt(dt_uint)) ∧
                                   (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
                         THEN ok_result(bl′)
                      ELSE fatal_result
                      ENDIF
                       ##
                       (λ (bl: list[Byte]):
                            access_count(a′, bl, cp′) ##
                             (λ (bl: list[Byte]):
                                  read_rnd_val(a′, bl, cp′) ##
                                   (λ (bl: list[Byte]):
                                        read_mem_cnt(a′, bl, cp′) ##
                                         (λ (bl: list[Byte]): read_seed(a′, bl, cp′))))))
                                (s′))
-------------------------------------------------------------------------------
{1}   state((IF null?(bl′) ∨
                    disjoint?(address_block(a′, length(bl′)), address_block(base, size)) ∨
                     (¬ cp′ ∧
                        length(bl′) = size(uidt(dt_uint)) ∧
                         (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
                  THEN ok_result(bl′)
               ELSE fatal_result
               ENDIF
                ##
                (λ (bl: list[Byte]):
                     access_count(a′, bl, cp′) ##
                      (λ (bl: list[Byte]):
                           read_rnd_val(a′, bl, cp′) ##
                            (λ (bl: list[Byte]):
                                 read_mem_cnt(a′, bl, cp′) ##
                                  (λ (bl: list[Byte]): read_seed(a′, bl, cp′))))))
                 (s′))‘state
      = s′‘state
```

Replacing using formula -1,
Expanding the definition of access_count,
Expanding the definition of read_rnd_val,
Expanding the definition of read_mem_cnt,
Expanding the definition of read_seed,
Expanding the definition of ##,
Expanding the definition of ok_result,

Expanding the definition of has_next_state,
Lifting IF-conditions to the top level,
Expanding the definition of increase_access_count,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `random_side_effects_unchanged_invariant.2.1.1`.
`random_side_effects_unchanged_invariant.2.1.2`:

```
{-1}  has_next_state((IF null?(bl′) ∨
                        disjoint?(address_block(a′, length(bl′)),
                                  address_block(base, size))
                       ∨
                       (¬ cp′ ∧
                          length(bl′) = size(uidt(dt_uint)) ∧
                            (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
                     THEN ok_result(bl′)
                     ELSE fatal_result
                     ENDIF
                      ##
                      (λ (bl: list[Byte]):
                           access_count(a′, bl, cp′) ##
                            (λ (bl: list[Byte]):
                                 read_rnd_val(a′, bl, cp′) ##
                                  (λ (bl: list[Byte]):
                                       read_mem_cnt(a′, bl, cp′) ##
                                        (λ (bl: list[Byte]): read_seed(a′, bl, cp′))))))
                     (s′))
{1}   null?(bl′) ∨
        disjoint?(address_block(a′, length(bl′)),
                  address_block(base, size[Physical_memory, pm_phy]))
       ∨
        (¬ cp′ ∧
           length(bl′) = size(uidt(dt_uint)) ∧
             (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
{2}   state((IF null?(bl′) ∨
                 disjoint?(address_block(a′, length(bl′)), address_block(base, size)) ∨
                  (¬ cp′ ∧
                     length(bl′) = size(uidt(dt_uint)) ∧
                       (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
               THEN ok_result(bl′)
               ELSE fatal_result
               ENDIF
                ##
                (λ (bl: list[Byte]):
                     access_count(a′, bl, cp′) ##
                      (λ (bl: list[Byte]):
                           read_rnd_val(a′, bl, cp′) ##
                            (λ (bl: list[Byte]):
                                 read_mem_cnt(a′, bl, cp′) ##
                                  (λ (bl: list[Byte]): read_seed(a′, bl, cp′))))))
               (s′))‘state
      = s′‘state
```

Replacing using formula 1,
Expanding the definition of fatal_result,

Expanding the definition of ##,

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of `random_side_effects_unchanged_invariant.2.1.2`.

`random_side_effects_unchanged_invariant.2.2:`

{-1}	has_next_state($q'(s')$)
{-2}	dwse_super_transformers((pm_phy`rw_addr \ address_block(base, size)), rnd_dev_write_side_effect) (q')
{1}	state($q'(s')$)`state = $s'$`state

Expanding the definition of dwse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 1) and hiding *,

Rewriting using has_next_state_expr_2_super, matching in *,

Rewriting using state_expr_2_super, matching in *,

Expanding the definition of rnd_dev_write_side_effect,

Expanding the definition of unaligned_access,

Case splitting on null?(bl!1) OR disjoint?(address_block(a!1, length(bl!1)), address_block(base, size[Physical_memory, pm_phy])) OR (NOT cp!1 AND length(bl!1) = size(uidt(dt_uint)) AND (a!1 = rnd_seed OR a!1 = mem_cnt OR a!1 = rnd_val)),

we get 2 subgoals:

`random_side_effects_unchanged_invariant.2.2.1:`

```
{-1}  null?(bl′) ∨
       disjoint?(address_block(a′, length(bl′)),
                 address_block(base, size[Physical_memory, pm_phy]))
        ∨
        (¬ cp′ ∧
           length(bl′) = size(uidt(dt_uint)) ∧
            (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
{-2}  has_next_state((IF null?(bl′) ∨
                        disjoint?(address_block(a′, length(bl′)),
                                  address_block(base, size))
                         ∨
                         (¬ cp′ ∧
                            length(bl′) = size(uidt(dt_uint)) ∧
                             (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
                       THEN ok_result(bl′)
                      ELSE fatal_result
                      ENDIF
                       ##
                       (λ (bl: list[Byte]):
                            access_count(a′, bl, cp′) ##
                             (λ (bl: list[Byte]):
                                  write_rnd_dev(a′, bl, cp′) ##
                                   (λ (bl: list[Byte]): reset_count(a′, bl, cp′)))))
                       (s′))
-------------------------------------------------------------------------------
{1}   state((IF null?(bl′) ∨
                 disjoint?(address_block(a′, length(bl′)), address_block(base, size)) ∨
                  (¬ cp′ ∧
                     length(bl′) = size(uidt(dt_uint)) ∧
                      (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
               THEN ok_result(bl′)
              ELSE fatal_result
              ENDIF
               ##
               (λ (bl: list[Byte]):
                    access_count(a′, bl, cp′) ##
                     (λ (bl: list[Byte]):
                          write_rnd_dev(a′, bl, cp′) ##
                           (λ (bl: list[Byte]): reset_count(a′, bl, cp′)))))
                (s′))‘state
      = s′‘state
```

Replacing using formula -1,
Expanding the definition of access_count,
Expanding the definition of write_rnd_dev,
Expanding the definition of increase_access_count,
Expanding the definition of reset_count,
Expanding the definition of ok_result,
Expanding the definition of ##,
Expanding the definition of clear_counter,
Expanding the definition of fatal_result,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_unchanged_invariant.2.2.1`.

`random_side_effects_unchanged_invariant.2.2.2`:

```
{-1}  has_next_state((IF null?(bl′) ∨
                        disjoint?(address_block(a′, length(bl′)),
                                  address_block(base, size))
                       ∨
                       (¬ cp′ ∧
                          length(bl′) = size(uidt(dt_uint)) ∧
                            (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
                     THEN ok_result(bl′)
                     ELSE fatal_result
                     ENDIF
                      ##
                      (λ (bl: list[Byte]):
                           access_count(a′, bl, cp′) ##
                            (λ (bl: list[Byte]):
                                 write_rnd_dev(a′, bl, cp′) ##
                                  (λ (bl: list[Byte]): reset_count(a′, bl, cp′)))))
                      (s′))
-------------------------------------------------------------------------------
{1}   null?(bl′) ∨
       disjoint?(address_block(a′, length(bl′)),
                 address_block(base, size[Physical_memory, pm_phy]))
        ∨
        (¬ cp′ ∧
           length(bl′) = size(uidt(dt_uint)) ∧
             (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
{2}   state((IF null?(bl′) ∨
                 disjoint?(address_block(a′, length(bl′)), address_block(base, size)) ∨
                  (¬ cp′ ∧
                     length(bl′) = size(uidt(dt_uint)) ∧
                       (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
               THEN ok_result(bl′)
               ELSE fatal_result
               ENDIF
                ##
                (λ (bl: list[Byte]):
                     access_count(a′, bl, cp′) ##
                      (λ (bl: list[Byte]):
                           write_rnd_dev(a′, bl, cp′) ##
                            (λ (bl: list[Byte]): reset_count(a′, bl, cp′)))))
                (s′))‘state
       = s′‘state
```

Replacing using formula 1,

Expanding the definition of fatal_result,

Expanding the definition of ##,

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of `random_side_effects_unchanged_invariant.2.2.2`.

Q.E.D.

C.23.5 Challenge_Random_Device.random_side_effects_transformers_ok

Terse proof for `random_side_effects_transformers_ok`.

`random_side_effects_transformers_ok:`

```
{1}  transformers_ok?(pm‘states,
                      (drse_super_transformers((pm‘ro_addr ∪ pm‘rw_addr), rnd_dev_read_side_effect) ∪ dwse_s
```

Installing automatic rewrites from: ok_expr_2_super has_next_state_expr_2_super fatal_result state_expr_2_super ok_result_state ok_result_ok expr_2_super_ok_result expr_2_super_fatal_result has_next_state_ok_result has_next_state_fatal_result ok_result_data in_blessed_memory_disjoint_device

Rewriting using transformers_ok_union_transformers, matching in *,

we get 2 subgoals:

`random_side_effects_transformers_ok.1:`

```
{1}  transformers_ok?(pm‘states,
                      drse_super_transformers((pm‘ro_addr ∪ pm‘rw_addr),
                                              rnd_dev_read_side_effect))
{2}  transformers_ok?(pm‘states,
                      (drse_super_transformers((pm‘ro_addr ∪ pm‘rw_addr), rnd_dev_read_side_effect) ∪ dwse_s
```

Rewriting using transformers_ok_all_transformers, matching in *,

Expanding the definition of drse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of rnd_dev_read_side_effect,

Using lemma in_blessed_memory_disjoint_device,

Using lemma in_blessed_memory_not_device_address,

Replacing using formula -3,

Case splitting on transformers_ok?(pm‘states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl!1)))),

we get 2 subgoals:

`random_side_effects_transformers_ok.1.1:`

```
{-1}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{-2}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-3}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                     access_count(a′, bl, cp′) ##
                                                       (λ (bl: list[Byte]):
                                                          read_rnd_val(a′, bl, cp′) ##
                                                            (λ (bl: list[Byte]):
                                                               read_mem_cnt(a′, bl, cp′)
                                                               ##
                                                               (λ
                                                                  (bl: list[Byte]):
                                                                  read_seed
                                                                  (a′, bl, cp′)))))))))
```

Case splitting on transformer_invariant?(pm‘states, singleton(expr_2_super (ok_result [Random_device_memory, list[Byte]] (bl!1)))),

we get 2 subgoals:

`random_side_effects_transformers_ok.1.1.1:`

```
{-1}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                             (bl′))))
{-2}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{-3}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-4}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                     access_count(a′, bl, cp′) ##
                                                       (λ (bl: list[Byte]):
                                                          read_rnd_val(a′, bl, cp′) ##
                                                            (λ (bl: list[Byte]):
                                                               read_mem_cnt(a′, bl, cp′)
                                                               ##
                                                               (λ
                                                                  (bl: list[Byte]):
                                                                  read_seed
                                                                  (a′, bl, cp′)))))))))
```

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.1.1.1.1:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-3}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-4}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-5}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm`states, singleton(expr_2_super(unaligned_access(a', bl', cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                  (λ (bl: list[Byte]):
                                                     read_rnd_val(a', bl, cp') ##
                                                     (λ (bl: list[Byte]):
                                                        read_mem_cnt(a', bl, cp')
                                                        ##
                                                        (λ
                                                          (bl: list[Byte]):
                                                          read_seed
                                                          (a', bl, cp')))))))))
```

Expanding the definition of unaligned_access,
which is trivially true.
This completes the proof of `random_side_effects_transformers_ok.1.1.1.1`.

`random_side_effects_transformers_ok.1.1.1.2:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-3}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-4}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-5}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm`states,
                             singleton(expr_2_super(unaligned_access(a', bl', cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                  (λ (bl: list[Byte]):
                                                     read_rnd_val(a', bl, cp') ##
                                                     (λ (bl: list[Byte]):
                                                        read_mem_cnt(a', bl, cp')
                                                        ##
                                                        (λ
                                                          (bl: list[Byte]):
                                                          read_seed
                                                          (a', bl, cp')))))))))
```

Expanding the definition of unaligned_access,

which is trivially true.

This completes the proof of `random_side_effects_transformers_ok.1.1.1.2`.

`random_side_effects_transformers_ok.1.1.1.3:`

```
{-1}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-2}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-3}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-4}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-5}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   ∀ (s: (pm‘states)):
        OK?(unaligned_access(a′, bl′, cp′)(s)) ⊃
          data(unaligned_access(a′, bl′, cp′)(s)) = bl′
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a′, bl, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                     read_rnd_val(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        read_mem_cnt(a′, bl, cp′)
                                                        ##
                                                        (λ
                                                          (bl: list[Byte]):
                                                          read_seed
                                                          (a′, bl, cp′)))))))))
```

Expanding the definition of unaligned_access,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.3`.

`random_side_effects_transformers_ok.1.1.1.4:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-3}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-4}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-5}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl')):
        transformers_ok?(pm`states,
                         singleton(expr_2_super(access_count(a', d, cp') ##
                                                (λ (bl: list[Byte]):
                                                   read_rnd_val(a', bl, cp') ##
                                                   (λ (bl: list[Byte]):
                                                      read_mem_cnt(a', bl, cp') ##
                                                      (λ (bl: list[Byte]):
                                                         read_seed(a', bl, cp')))))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                  (λ (bl: list[Byte]):
                                                     read_rnd_val(a', bl, cp') ##
                                                     (λ (bl: list[Byte]):
                                                        read_mem_cnt(a', bl, cp')
                                                        ##
                                                        (λ
                                                          (bl: list[Byte]):
                                                          read_seed
                                                          (a', bl, cp')))))))))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.1.1.1.4.1:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-5}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-6}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states, singleton(expr_2_super(access_count(a′, d′, cp′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 read_rnd_val(a′, bl, cp′) ##
                                                 (λ (bl: list[Byte]):
                                                    read_mem_cnt(a′, bl, cp′) ##
                                                    (λ (bl: list[Byte]):
                                                       read_seed(a′, bl, cp′)))))))
```

Expanding the definition of access_count,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.1`.

`random_side_effects_transformers_ok.1.1.1.4.2:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-5}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-6}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states, singleton(expr_2_super(access_count(a′, d′, cp′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 read_rnd_val(a′, bl, cp′) ##
                                                 (λ (bl: list[Byte]):
                                                    read_mem_cnt(a′, bl, cp′) ##
                                                    (λ (bl: list[Byte]):
                                                       read_seed(a′, bl, cp′)))))))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.2`.

`random_side_effects_transformers_ok.1.1.1.4.3:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-5}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-6}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   ∀ (s: (pm‘states)):
        OK?(access_count(a′, d′, cp′)(s)) ⊃
          data(access_count(a′, d′, cp′)(s)) = bl′
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 read_rnd_val(a′, bl, cp′) ##
                                                   (λ (bl: list[Byte]):
                                                      read_mem_cnt(a′, bl, cp′) ##
                                                        (λ (bl: list[Byte]):
                                                           read_seed(a′, bl, cp′)))))))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of access_count,
which is trivially true.
This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.3`.
`random_side_effects_transformers_ok.1.1.1.4.4:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-5}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-6}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl′)):
        transformers_ok?(pm‘states,
                         singleton(expr_2_super(read_rnd_val(a′, d, cp′) ##
                                                (λ (bl: list[Byte]):
                                                   read_mem_cnt(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        read_seed(a′, bl, cp′))))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 read_rnd_val(a′, bl, cp′) ##
                                                   (λ (bl: list[Byte]):
                                                      read_mem_cnt(a′, bl, cp′) ##
                                                        (λ (bl: list[Byte]):
                                                           read_seed(a′, bl, cp′)))))))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.1.1.1.4.4.1:`

{-1} $d'' = \mathrm{bl}'$
{-2} $d' = \mathrm{bl}'$
{-3} transformer_invariant?(pm‘states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]](bl′))))
{-4} transformers_ok?(pm‘states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]](bl′))))
{-5} cons?(bl′) ⊃ $a' \neq$ rnd_seed ∧ $a' \neq$ mem_cnt ∧ $a' \neq$ rnd_val
{-6} disjoint?(address_block(a', length(bl′)), address_block(base, size))
{-7} (address_block(a', length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))

{1} transformers_ok?(pm‘states, singleton(expr_2_super(read_rnd_val(a', d'', cp′))))
{2} transformers_ok?(pm‘states, singleton(expr_2_super(read_rnd_val(a', d'', cp′) ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp′) ## (λ (bl: list[Byte]): read_seed(a', bl, cp′))))))

Hiding formulas: 2,

Expanding the definition of read_rnd_val,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.1`.

`random_side_effects_transformers_ok.1.1.1.4.4.2:`

{-1} $d'' = \mathrm{bl}'$
{-2} $d' = \mathrm{bl}'$
{-3} transformer_invariant?(pm‘states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]](bl′))))
{-4} transformers_ok?(pm‘states, singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]](bl′))))
{-5} cons?(bl′) ⊃ $a' \neq$ rnd_seed ∧ $a' \neq$ mem_cnt ∧ $a' \neq$ rnd_val
{-6} disjoint?(address_block(a', length(bl′)), address_block(base, size))
{-7} (address_block(a', length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))

{1} transformer_invariant?(pm‘states, singleton(expr_2_super(read_rnd_val(a', d'', cp′))))
{2} transformers_ok?(pm‘states, singleton(expr_2_super(read_rnd_val(a', d'', cp′) ## (λ (bl: list[Byte]): read_mem_cnt(a', bl, cp′) ## (λ (bl: list[Byte]): read_seed(a', bl, cp′))))))

Expanding the definition of read_rnd_val,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.2`.

`random_side_effects_transformers_ok.1.1.1.4.4.3:`

```
{-1}  d'' = bl'
{-2}  d' = bl'
{-3}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-4}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-5}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-6}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-7}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
-------------------------------------------------------------------
{1}   ∀ (s: (pm`states)):
        OK?(read_rnd_val(a', d'', cp')(s)) ⊃
          data(read_rnd_val(a', d'', cp')(s)) = bl'
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(read_rnd_val(a', d'', cp') ##
                                              (λ (bl: list[Byte]):
                                                 read_mem_cnt(a', bl, cp') ##
                                                   (λ (bl: list[Byte]):
                                                      read_seed(a', bl, cp'))))))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of read_rnd_val,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.3`.
`random_side_effects_transformers_ok.1.1.1.4.4.4:`

```
{-1}  d'' = bl'
{-2}  d' = bl'
{-3}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-4}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-5}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-6}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-7}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
-------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl')):
        transformers_ok?(pm`states,
                         singleton(expr_2_super(read_mem_cnt(a', d, cp') ##
                                                (λ (bl: list[Byte]):
                                                   read_seed(a', bl, cp')))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(read_rnd_val(a', d'', cp') ##
                                              (λ (bl: list[Byte]):
                                                 read_mem_cnt(a', bl, cp') ##
                                                   (λ (bl: list[Byte]):
                                                      read_seed(a', bl, cp'))))))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: (-1 2),
Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl:

list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.1.1.1.4.4.4.1:`

```
{-1}  d''' = bl'
{-2}  d'' = bl'
{-3}  d' = bl'
{-4}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-5}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-6}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-7}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-8}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm`states, singleton(expr_2_super(read_mem_cnt(a', d''', cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(read_mem_cnt(a', d''', cp') ##
                                              (λ (bl: list[Byte]):
                                                 read_seed(a', bl, cp')))))
```

Expanding the definition of read_mem_cnt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.4.1`.

`random_side_effects_transformers_ok.1.1.1.4.4.4.2:`

```
{-1}  d''' = bl'
{-2}  d'' = bl'
{-3}  d' = bl'
{-4}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-5}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-6}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-7}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-8}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm`states, singleton(expr_2_super(read_mem_cnt(a', d''', cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(read_mem_cnt(a', d''', cp') ##
                                              (λ (bl: list[Byte]):
                                                 read_seed(a', bl, cp')))))
```

Expanding the definition of read_mem_cnt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.4.2`.

`random_side_effects_transformers_ok.1.1.1.4.4.4.3:`

```
{-1}  d''' = bl'
{-2}  d'' = bl'
{-3}  d' = bl'
{-4}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-5}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-6}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-7}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-8}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   ∀ (s: (pm`states)):
        OK?(read_mem_cnt(a', d''', cp')(s)) ⊃
          data(read_mem_cnt(a', d''', cp')(s)) = bl'
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(read_mem_cnt(a', d''', cp') ##
                                              (λ (bl: list[Byte]):
                                                 read_seed(a', bl, cp')))))
```

Expanding the definition of read_mem_cnt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.4.3`.

`random_side_effects_transformers_ok.1.1.1.4.4.4.4:`

```
{-1}  d''' = bl'
{-2}  d'' = bl'
{-3}  d' = bl'
{-4}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-5}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-6}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-7}  disjoint?(address_block(a', length(bl')), address_block(base, size))
{-8}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl')):
        transformers_ok?(pm`states, singleton(expr_2_super(read_seed(a', d, cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(read_mem_cnt(a', d''', cp') ##
                                              (λ (bl: list[Byte]):
                                                 read_seed(a', bl, cp')))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of read_seed,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.1.4.4.4.4`.

`random_side_effects_transformers_ok.1.1.2:`

```
{-1}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{-2}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-3}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                             (bl′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a′, bl, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                     read_rnd_val(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        read_mem_cnt(a′, bl, cp′)
                                                        ##
                                                        (λ
                                                           (bl: list[Byte]):
                                                           read_seed
                                                           (a′, bl, cp′)))))))))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.1.2`.

`random_side_effects_transformers_ok.1.2:`

```
{-1}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-2}  disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a′, bl, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                     read_rnd_val(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        read_mem_cnt(a′, bl, cp′)
                                                        ##
                                                        (λ
                                                           (bl: list[Byte]):
                                                           read_seed
                                                           (a′, bl, cp′)))))))))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.1.2`.

`random_side_effects_transformers_ok.2:`

```
{1}  transformers_ok?(pm`states, dwse_super_transformers(pm`rw_addr, rnd_dev_write_side_effect))
{2}  transformers_ok?(pm`states,
                      (drse_super_transformers((pm`ro_addr ∪ pm`rw_addr), rnd_dev_read_side_effect) ∪ dwse_s
```

Rewriting using transformers_ok_all_transformers, matching in *,

Expanding the definition of dwse_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of rnd_dev_write_side_effect,

Using lemma in_blessed_memory_disjoint_device,

Using lemma in_blessed_memory_not_device_address,

Case splitting on subset?(address_block(a!1, length(bl!1)), union(pm`ro_addr, pm`rw_addr)),

we get 2 subgoals:

`random_side_effects_transformers_ok.2.1:`

```
{-1}  (address_block(a′, length(bl′)) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-2}  (address_block(a′, length(bl′)) ⊆ (pm`ro_addr ∪ pm`rw_addr)) ∧ cons?(bl′) ⊃
       a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-3}  (address_block(a′, length(bl′)) ⊆ (pm`ro_addr ∪ pm`rw_addr)) ⊃
       disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-4}  (address_block(a′, length(bl′)) ⊆ pm`rw_addr)
-------------------------------------------------------------------------------------------------
{1}  transformers_ok?(pm`states,
                      singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                (λ (bl: list[Byte]):
                                                   access_count(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        write_rnd_dev(a′, bl, cp′) ##
                                                          (λ (bl: list[Byte]):
                                                             reset_count
                                                             (a′, bl, cp′)))))))))
```

Case splitting on transformers_ok?(pm`states, singleton(expr_2_super(ok_result [Random_device_memory, list[Byte]] (bl!1)))),

we get 2 subgoals:

`random_side_effects_transformers_ok.2.1.1:`

```
{-1}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{-2}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ cons?(bl′) ⊃
       a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
       disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-5}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                       access_count(a′, bl, cp′) ##
                                                        (λ (bl: list[Byte]):
                                                             write_rnd_dev(a′, bl, cp′) ##
                                                              (λ (bl: list[Byte]):
                                                                   reset_count
                                                                   (a′, bl, cp′)))))))))
```

Case splitting on transformer_invariant?(pm‘states, singleton(expr_2_super (ok_result [Random_device_memory, list[Byte]] (bl!1)))),

we get 2 subgoals:

`random_side_effects_transformers_ok.2.1.1.1:`

```
{-1}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]
                                                             (bl′))))
{-2}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ cons?(bl′) ⊃
       a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-5}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
       disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-6}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                       access_count(a′, bl, cp′) ##
                                                        (λ (bl: list[Byte]):
                                                             write_rnd_dev(a′, bl, cp′) ##
                                                              (λ (bl: list[Byte]):
                                                                   reset_count
                                                                   (a′, bl, cp′)))))))))
```

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.2.1.1.1.1:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-3}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-4}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-5}  TRUE ⊃ TRUE
{-6}  (address_block(a', length(bl')) ⊆ pm`rw_addr)
------------------------------------------------------------------------
{1}   transformers_ok?(pm`states, singleton(expr_2_super(unaligned_access(a', bl', cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                  (λ (bl: list[Byte]):
                                                     write_rnd_dev(a', bl, cp') ##
                                                     (λ (bl: list[Byte]):
                                                        reset_count
                                                        (a', bl, cp')))))))))
```

Expanding the definition of unaligned_access,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.1`.

`random_side_effects_transformers_ok.2.1.1.1.2:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-3}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-4}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-5}  TRUE ⊃ TRUE
{-6}  (address_block(a', length(bl')) ⊆ pm`rw_addr)
------------------------------------------------------------------------
{1}   transformer_invariant?(pm`states,
                             singleton(expr_2_super(unaligned_access(a', bl', cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                  (λ (bl: list[Byte]):
                                                     write_rnd_dev(a', bl, cp') ##
                                                     (λ (bl: list[Byte]):
                                                        reset_count
                                                        (a', bl, cp')))))))))
```

Expanding the definition of unaligned_access,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.2`.

`random_side_effects_transformers_ok.2.1.1.1.3:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-3}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-4}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-5}  TRUE ⊃ TRUE
{-6}  (address_block(a', length(bl')) ⊆ pm`rw_addr)
-------------------------------------------------------------------------------
{1}   ∀ (s: (pm`states)):
        OK?(unaligned_access(a', bl', cp')(s)) ⊃
          data(unaligned_access(a', bl', cp')(s)) = bl'
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                    (λ (bl: list[Byte]):
                                                       write_rnd_dev(a', bl, cp') ##
                                                         (λ (bl: list[Byte]):
                                                            reset_count
                                                            (a', bl, cp'))))))))))
```

Expanding the definition of unaligned_access,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.3`.

`random_side_effects_transformers_ok.2.1.1.1.4:`

```
{-1}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-2}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                        (bl'))))
{-3}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-4}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-5}  TRUE ⊃ TRUE
{-6}  (address_block(a', length(bl')) ⊆ pm`rw_addr)
-------------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl')):
        transformers_ok?(pm`states,
                         singleton(expr_2_super(access_count(a', d, cp') ##
                                                (λ (bl: list[Byte]):
                                                   write_rnd_dev(a', bl, cp') ##
                                                   (λ (bl: list[Byte]):
                                                      re-
      set_count(a', bl, cp'))))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super((unaligned_access(a', bl', cp') ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a', bl, cp') ##
                                                  (λ (bl: list[Byte]):
                                                     write_rnd_dev(a', bl, cp') ##
                                                     (λ (bl: list[Byte]):
                                                        reset_count
                                                        (a', bl, cp'))))))))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.2.1.1.1.4.1:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-5}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-6}  TRUE ⊃ TRUE
{-7}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states, singleton(expr_2_super(access_count(a′, d′, cp′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 write_rnd_dev(a′, bl, cp′) ##
                                                 (λ (bl: list[Byte]):
                                                    re-
      set_count(a′, bl, cp′))))))
```

Expanding the definition of access_count,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.1`.

`random_side_effects_transformers_ok.2.1.1.1.4.2:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-5}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-6}  TRUE ⊃ TRUE
{-7}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states, singleton(expr_2_super(access_count(a′, d′, cp′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 write_rnd_dev(a′, bl, cp′) ##
                                                 (λ (bl: list[Byte]):
                                                    re-
      set_count(a′, bl, cp′))))))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.2`.

`random_side_effects_transformers_ok.2.1.1.1.4.3:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-5}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-6}  TRUE ⊃ TRUE
{-7}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------------
{1}   ∀ (s: (pm‘states)):
        OK?(access_count(a′, d′, cp′)(s)) ⊃
          data(access_count(a′, d′, cp′)(s)) = bl′
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 write_rnd_dev(a′, bl, cp′) ##
                                                   (λ (bl: list[Byte]):
                                                      re-
      set_count(a′, bl, cp′))))))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of access_count,
which is trivially true.
This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.3`.
`random_side_effects_transformers_ok.2.1.1.1.4.4:`

```
{-1}  d′ = bl′
{-2}  transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl′))))
{-3}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl′))))
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-5}  cons?(bl′) ⊃ a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-6}  TRUE ⊃ TRUE
{-7}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl′)):
        transformers_ok?(pm‘states,
                         singleton(expr_2_super(write_rnd_dev(a′, d, cp′) ##
                                                (λ (bl: list[Byte]):
                                                   reset_count(a′, bl, cp′)))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(access_count(a′, d′, cp′) ##
                                              (λ (bl: list[Byte]):
                                                 write_rnd_dev(a′, bl, cp′) ##
                                                   (λ (bl: list[Byte]):
                                                      re-
      set_count(a′, bl, cp′))))))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: (-1 2),

Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (bl: list[Byte]): bl = bl!1,

we get 4 subgoals:

`random_side_effects_transformers_ok.2.1.1.1.4.4.1:`

{-1} $d'' = \mathrm{bl}'$
{-2} $d' = \mathrm{bl}'$
{-3} transformer_invariant?(pm‘states,
singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
(bl′))))
{-4} transformers_ok?(pm‘states,
singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
(bl′))))
{-5} (address_block(a', length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-6} cons?(bl′) ⊃ $a' \neq$ rnd_seed ∧ $a' \neq$ mem_cnt ∧ $a' \neq$ rnd_val
{-7} TRUE ⊃ TRUE
{-8} (address_block(a', length(bl′)) ⊆ pm‘rw_addr)

{1} transformers_ok?(pm‘states, singleton(expr_2_super(write_rnd_dev(a', d'', cp′))))
{2} transformers_ok?(pm‘states,
singleton(expr_2_super(write_rnd_dev(a', d'', cp′) ##
(λ (bl: list[Byte]):
reset_count(a', bl, cp′)))))

Hiding formulas: 2,

Expanding the definition of write_rnd_dev,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.4.1`.

`random_side_effects_transformers_ok.2.1.1.1.4.4.2:`

{-1} $d'' = \mathrm{bl}'$
{-2} $d' = \mathrm{bl}'$
{-3} transformer_invariant?(pm‘states,
singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
(bl′))))
{-4} transformers_ok?(pm‘states,
singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
(bl′))))
{-5} (address_block(a', length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-6} cons?(bl′) ⊃ $a' \neq$ rnd_seed ∧ $a' \neq$ mem_cnt ∧ $a' \neq$ rnd_val
{-7} TRUE ⊃ TRUE
{-8} (address_block(a', length(bl′)) ⊆ pm‘rw_addr)

{1} transformer_invariant?(pm‘states, singleton(expr_2_super(write_rnd_dev(a', d'', cp′))))
{2} transformers_ok?(pm‘states,
singleton(expr_2_super(write_rnd_dev(a', d'', cp′) ##
(λ (bl: list[Byte]):
reset_count(a', bl, cp′)))))

Hiding formulas: 2,

Expanding the definition of write_rnd_dev,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.4.2`.

`random_side_effects_transformers_ok.2.1.1.1.4.4.3:`

```
{-1}  d'' = bl'
{-2}  d' = bl'
{-3}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-4}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-5}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-6}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-7}  TRUE ⊃ TRUE
{-8}  (address_block(a', length(bl')) ⊆ pm`rw_addr)
-------------------------------------------------------------------------
{1}   ∀ (s: (pm`states)):
        OK?(write_rnd_dev(a', d'', cp')(s)) ⊃
          data(write_rnd_dev(a', d'', cp')(s)) = bl'
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(write_rnd_dev(a', d'', cp') ##
                                              (λ (bl: list[Byte]):
                                                 reset_count(a', bl, cp')))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of write_rnd_dev,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.4.3`.

`random_side_effects_transformers_ok.2.1.1.1.4.4.4:`

```
{-1}  d'' = bl'
{-2}  d' = bl'
{-3}  transformer_invariant?(pm`states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                    (bl'))))
{-4}  transformers_ok?(pm`states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                              (bl'))))
{-5}  (address_block(a', length(bl')) ⊆ (pm`ro_addr ∪ pm`rw_addr))
{-6}  cons?(bl') ⊃ a' ≠ rnd_seed ∧ a' ≠ mem_cnt ∧ a' ≠ rnd_val
{-7}  TRUE ⊃ TRUE
{-8}  (address_block(a', length(bl')) ⊆ pm`rw_addr)
-------------------------------------------------------------------------
{1}   ∀ (d: (λ (bl: list[Byte]): bl = bl')):
        transformers_ok?(pm`states, singleton(expr_2_super(reset_count(a', d, cp'))))
{2}   transformers_ok?(pm`states,
                       singleton(expr_2_super(write_rnd_dev(a', d'', cp') ##
                                              (λ (bl: list[Byte]):
                                                 reset_count(a', bl, cp')))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of reset_count,

Hiding formulas: (-1 2 -9),

Replacing using formula -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.1.4.4.4`.

`random_side_effects_transformers_ok.2.1.1.2:`

```
{-1}  transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{-2}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-3}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ cons?(bl′) ⊃
       a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-4}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
       disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-5}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states,
                             singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                             (bl′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                 (λ (bl: list[Byte]):
                                                    access_count(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        write_rnd_dev(a′, bl, cp′) ##
                                                         (λ (bl: list[Byte]):
                                                            reset_count
                                                            (a′, bl, cp′)))))))))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.1.2`.

`random_side_effects_transformers_ok.2.1.2:`

```
{-1}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{-2}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ cons?(bl′) ⊃
       a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-3}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
       disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-4}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------------
{1}   transformers_ok?(pm‘states,
                       singleton(expr_2_super(ok_result[Random_device_memory, list[Byte]]
                                                       (bl′))))
{2}   transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                                 (λ (bl: list[Byte]):
                                                    access_count(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        write_rnd_dev(a′, bl, cp′) ##
                                                         (λ (bl: list[Byte]):
                                                            reset_count
                                                            (a′, bl, cp′)))))))))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_transformers_ok.2.1.2`.

`random_side_effects_transformers_ok.2.2:`

```
{-1}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ cons?(bl′) ⊃
       a′ ≠ rnd_seed ∧ a′ ≠ mem_cnt ∧ a′ ≠ rnd_val
{-2}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
       disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{-3}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
------------------------------------------------------------------------------
 {1}  (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
 {2}  transformers_ok?(pm‘states,
                       singleton(expr_2_super((unaligned_access(a′, bl′, cp′) ##
                                               (λ (bl: list[Byte]):
                                                  access_count(a′, bl, cp′) ##
                                                  (λ (bl: list[Byte]):
                                                     write_rnd_dev(a′, bl, cp′) ##
                                                     (λ (bl: list[Byte]):
                                                        reset_count
                                                        (a′, bl, cp′)))))))))
```

Keeping (-3 1) and hiding *,

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of `random_side_effects_transformers_ok.2.2`.

Q.E.D.

C.23.6 Challenge_Random_Device.random_side_effects_side_effect_content_unchanged_read

Terse proof for `random_side_effects_side_effect_content_unchanged_read`.

`random_side_effects_side_effect_content_unchanged_read:`

```
------------------------------------------------------------------------------
 {1}  side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, rnd_dev_read_side_effect)
```

Case splitting on size(uidt(dt_uint)) > 0,

we get 2 subgoals:

`random_side_effects_side_effect_content_unchanged_read.1:`

```
{-1}  size(uidt(dt_uint)) > 0
------------------------------------------------------------------------------
 {1}  side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, rnd_dev_read_side_effect)
```

Installing automatic rewrites from: ok_expr_2_super has_next_state_expr_2_super fatal_result state_expr_2_super ok_result_state ok_result_ok expr_2_super_ok_result expr_2_super_fatal_result has_next_state_ok_result has_next_state_fatal_result ok_result_data ertimes_ax

Expanding the definition of rnd_dev_read_side_effect,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

`random_side_effects_side_effect_content_unchanged_read.1.1:`

```
{-1}  size(uidt(dt_uint)) > 0
-------------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states,
                             se_super_transformers((pm‘ro_addr ∪ pm‘rw_addr), un-
      aligned_access))
{2}   side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      (unaligned_access(a, bl, cp) ##
                                         (λ (bl: list[Byte]):
                                            access_count(a, bl, cp) ##
                                             (λ (bl: list[Byte]):
                                                read_rnd_val(a, bl, cp) ##
                                                 (λ (bl: list[Byte]):
                                                    read_mem_cnt(a, bl, cp) ##
                                                     (λ (bl: list[Byte]):
                                                        read_seed(a, bl, cp)))))))
```

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.1`.

`random_side_effects_side_effect_content_unchanged_read.1.2:`

```
{-1}  size(uidt(dt_uint)) > 0
-------------------------------------------------------------------------------
{1}   side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, unaligned_access)
{2}   side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      (unaligned_access(a, bl, cp) ##
                                         (λ (bl: list[Byte]):
                                            access_count(a, bl, cp) ##
                                             (λ (bl: list[Byte]):
                                                read_rnd_val(a, bl, cp) ##
                                                 (λ (bl: list[Byte]):
                                                    read_mem_cnt(a, bl, cp) ##
                                                     (λ (bl: list[Byte]):
                                                        read_seed(a, bl, cp)))))))
```

Hiding formulas: 2,

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.2`.

`random_side_effects_side_effect_content_unchanged_read.1.3:`

```
{-1}  size(uidt(dt_uint)) > 0
{1}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      access_count(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            read_rnd_val(a, bl, cp) ##
                                             (λ (bl: list[Byte]):
                                                  read_mem_cnt(a, bl, cp) ##
                                                   (λ (bl: list[Byte]):
                                                        read_seed(a, bl, cp)))))
{2}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      (unaligned_access(a, bl, cp) ##
                                          (λ (bl: list[Byte]):
                                               access_count(a, bl, cp) ##
                                                (λ (bl: list[Byte]):
                                                     read_rnd_val(a, bl, cp) ##
                                                      (λ (bl: list[Byte]):
                                                           read_mem_cnt(a, bl, cp) ##
                                                            (λ (bl: list[Byte]):
                                                                 read_seed(a, bl, cp)))))))
```

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

`random_side_effects_side_effect_content_unchanged_read.1.3.1:`

```
{-1}  size(uidt(dt_uint)) > 0
{1}   transformer_invariant?(pm`states,
                             se_super_transformers((pm`ro_addr ∪ pm`rw_addr), ac-
      cess_count))
{2}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      access_count(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            read_rnd_val(a, bl, cp) ##
                                             (λ (bl: list[Byte]):
                                                  read_mem_cnt(a, bl, cp) ##
                                                   (λ (bl: list[Byte]):
                                                        read_seed(a, bl, cp)))))
```

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.1`.

`random_side_effects_side_effect_content_unchanged_read.1.3.2:`

{-1} size(uidt(dt_uint)) > 0

{1} side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, access_count)

{2} side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
λ (a: Address, bl: list[Byte], cp: bool):
access_count(a, bl, cp) ##
(λ (bl: list[Byte]):
read_rnd_val(a, bl, cp) ##
(λ (bl: list[Byte]):
read_mem_cnt(a, bl, cp) ##
(λ (bl: list[Byte]):
read_seed(a, bl, cp)))))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.2`.
`random_side_effects_side_effect_content_unchanged_read.1.3.3:`

{-1} size(uidt(dt_uint)) > 0

{1} side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
λ (a: Address, bl: list[Byte], cp: bool):
read_rnd_val(a, bl, cp) ##
(λ (bl: list[Byte]):
read_mem_cnt(a, bl, cp) ##
(λ (bl: list[Byte]): read_seed(a, bl, cp))))

{2} side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
λ (a: Address, bl: list[Byte], cp: bool):
access_count(a, bl, cp) ##
(λ (bl: list[Byte]):
read_rnd_val(a, bl, cp) ##
(λ (bl: list[Byte]):
read_mem_cnt(a, bl, cp) ##
(λ (bl: list[Byte]):
read_seed(a, bl, cp)))))

Hiding formulas: 2,
Rewriting using side_effect_content_unchanged_composition, matching in *,
we get 3 subgoals:
`random_side_effects_side_effect_content_unchanged_read.1.3.3.1:`

{-1} size(uidt(dt_uint)) > 0

{1} transformer_invariant?(pm‘states,
se_super_transformers((pm‘ro_addr ∪ pm‘rw_addr), read_rnd_val))

{2} side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states,
λ (a: Address, bl: list[Byte], cp: bool):
read_rnd_val(a, bl, cp) ##
(λ (bl: list[Byte]):
read_mem_cnt(a, bl, cp) ##
(λ (bl: list[Byte]): read_seed(a, bl, cp))))

Rewriting using transformer_invariant_all_transformers, matching in *,
Expanding the definition of se_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.3.1`.
`random_side_effects_side_effect_content_unchanged_read.1.3.3.2`:

```
{-1}  size(uidt(dt_uint)) > 0
{1}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states, read_rnd_val)
{2}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      read_rnd_val(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            read_mem_cnt(a, bl, cp) ##
                                             (λ (bl: list[Byte]): read_seed(a, bl, cp))))
```

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.3.2`.
`random_side_effects_side_effect_content_unchanged_read.1.3.3.3`:

```
{-1}  size(uidt(dt_uint)) > 0
{1}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      read_mem_cnt(a, bl, cp) ##
                                       (λ (bl: list[Byte]): read_seed(a, bl, cp)))
{2}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      read_rnd_val(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            read_mem_cnt(a, bl, cp) ##
                                             (λ (bl: list[Byte]): read_seed(a, bl, cp))))
```

Hiding formulas: 2,
Rewriting using side_effect_content_unchanged_composition, matching in *,
we get 3 subgoals:
`random_side_effects_side_effect_content_unchanged_read.1.3.3.3.1`:

```
{-1}  size(uidt(dt_uint)) > 0
{1}   transformer_invariant?(pm`states,
                             se_super_transformers((pm`ro_addr ∪ pm`rw_addr), read_mem_cnt))
{2}   side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      read_mem_cnt(a, bl, cp) ##
                                       (λ (bl: list[Byte]): read_seed(a, bl, cp)))
```

Rewriting using transformer_invariant_all_transformers, matching in *,
Expanding the definition of se_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.3.3.1`.

`random_side_effects_side_effect_content_unchanged_read.1.3.3.3.2:`

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, read_mem_cnt)
{2}	side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, λ (a: Address, bl: list[Byte], cp: bool): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp)))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.3.3.2`.
`random_side_effects_side_effect_content_unchanged_read.1.3.3.3.3:`

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, read_seed)
{2}	side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, λ (a: Address, bl: list[Byte], cp: bool): read_mem_cnt(a, bl, cp) ## (λ (bl: list[Byte]): read_seed(a, bl, cp)))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.1.3.3.3.3`.
`random_side_effects_side_effect_content_unchanged_read.2:`

{1}	size(uidt(dt_uint)) > 0
{2}	side_effect_content_unchanged((pm‘ro_addr ∪ pm‘rw_addr), pm‘states, rnd_dev_read_side_effect)

Using lemma dt_uint_size,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_read.2`.
Q.E.D.

C.23.7 Challenge_Random_Device.random_side_effects_side_effect_content_unchanged_w

Terse proof for `random_side_effects_side_effect_content_unchanged_write`.
`random_side_effects_side_effect_content_unchanged_write:`

{1}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, rnd_dev_write_side_effect)

Case splitting on size(uidt(dt_uint)) > 0,
we get 2 subgoals:
`random_side_effects_side_effect_content_unchanged_write.1:`

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, rnd_dev_write_side_effect)

Installing automatic rewrites from: ok_expr_2_super has_next_state_expr_2_super fatal_result state_expr_2_super ok_result_state ok_result_ok expr_2_super_ok_result expr_2_super_fatal_result has_next_state_ok_result has_next_state_fatal_result ok_result_data ertimes_ax
Expanding the definition of rnd_dev_write_side_effect,
Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

`random_side_effects_side_effect_content_unchanged_write.1.1:`

```
{-1}  size(uidt(dt_uint)) > 0
-------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states, se_super_transformers(pm‘rw_addr, un-
      aligned_access))
{2}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      (unaligned_access(a, bl, cp) ##
                                         (λ (bl: list[Byte]):
                                            access_count(a, bl, cp) ##
                                              (λ (bl: list[Byte]):
                                                 write_rnd_dev(a, bl, cp) ##
                                                   (λ (bl: list[Byte]):
                                                      reset_count(a, bl, cp))))))
```

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

we get 2 subgoals:

`random_side_effects_side_effect_content_unchanged_write.1.1.1:`

```
{-1}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states, singleton(expr_2_super(ok_result(bl′))))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.1.1`.

`random_side_effects_side_effect_content_unchanged_write.1.1.2:`

```
{-1}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
-------------------------------------------------------------------------
{1}   (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{2}   transformer_invariant?(pm‘states,
                             IF null?(bl′) ∨
                                disjoint?(address_block(a′, length(bl′)),
                                          address_block(base, size))
                                ∨
                                (¬ cp′ ∧
                                   length(bl′) = size(uidt(dt_uint)) ∧
                                     (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
                               THEN singleton(expr_2_super(ok_result(bl′)))
                             ELSE singleton(expr_2_super(fatal_result))
                             ENDIF)
```

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.1.2`.

`random_side_effects_side_effect_content_unchanged_write.1.2:`

```
{-1}  size(uidt(dt_uint)) > 0
------------------------------------------------------------------------
{1}   side_effect_content_unchanged(pm‘rw_addr, pm‘states, unaligned_access)
{2}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      (unaligned_access(a, bl, cp) ##
                                         (λ (bl: list[Byte]):
                                            access_count(a, bl, cp) ##
                                              (λ (bl: list[Byte]):
                                                 write_rnd_dev(a, bl, cp) ##
                                                   (λ (bl: list[Byte]):
                                                      reset_count(a, bl, cp))))))
```

Hiding formulas: 2,

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Expanding the definition of unaligned_access,

Rewriting using in_blessed_memory_disjoint_device, matching in *,

we get 2 subgoals:

`random_side_effects_side_effect_content_unchanged_write.1.2.1:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                  real_pred(x) ∧
                    rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (bl′)
{-2}  pm‘states(s′)
{-3}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
{-4}  OK?(ok_result(bl′)(s′))
{-5}  size(uidt(dt_uint)) > 0
------------------------------------------------------------------------
{1}   data(ok_result(bl′)(s′)) = bl′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.2.1`.

`random_side_effects_side_effect_content_unchanged_write.1.2.2:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-2}  pm‘states(s′)
{-3}  (address_block(a′, length(bl′)) ⊆ pm‘rw_addr)
{-4}  OK?(IF null?(bl′) ∨
                disjoint?(address_block(a′, length(bl′)), address_block(base, size)) ∨
                 (¬ cp′ ∧
                     length(bl′) = size(uidt(dt_uint)) ∧
                      (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
              THEN ok_result(bl′)(s′)
            ELSE Fatal
            ENDIF)
{-5}  size(uidt(dt_uint)) > 0
──────────────────────────────────────────────────────────────────────
{1}   (address_block(a′, length(bl′)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
{2}   IF null?(bl′) ∨
         disjoint?(address_block(a′, length(bl′)), address_block(base, size)) ∨
          (¬ cp′ ∧
              length(bl′) = size(uidt(dt_uint)) ∧
               (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
        THEN bl′
      ELSE data(Fatal)
      ENDIF
       = bl′
```

Rewriting using subset_bigger_union_right, matching in *,
This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.2.2`.
`random_side_effects_side_effect_content_unchanged_write.1.3:`

```
{-1}  size(uidt(dt_uint)) > 0
──────────────────────────────────────────────────────────────────────
{1}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      access_count(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            write_rnd_dev(a, bl, cp) ##
                                             (λ (bl: list[Byte]): re-
      set_count(a, bl, cp))))
{2}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      (unaligned_access(a, bl, cp) ##
                                          (λ (bl: list[Byte]):
                                               access_count(a, bl, cp) ##
                                                (λ (bl: list[Byte]):
                                                     write_rnd_dev(a, bl, cp) ##
                                                      (λ (bl: list[Byte]):
                                                           reset_count(a, bl, cp))))))
```

Hiding formulas: 2,
Rewriting using side_effect_content_unchanged_composition, matching in *,
we get 3 subgoals:

`random_side_effects_side_effect_content_unchanged_write.1.3.1:`

```
{-1}  size(uidt(dt_uint)) > 0
------------------------------------------------------------------------
{1}   transformer_invariant?(pm‘states, se_super_transformers(pm‘rw_addr, access_count))
{2}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      access_count(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            write_rnd_dev(a, bl, cp) ##
                                             (λ (bl: list[Byte]): re-
      set_count(a, bl, cp))))
```

Rewriting using transformer_invariant_all_transformers, matching in *,

Expanding the definition of se_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.3.1`.

`random_side_effects_side_effect_content_unchanged_write.1.3.2:`

```
{-1}  size(uidt(dt_uint)) > 0
------------------------------------------------------------------------
{1}   side_effect_content_unchanged(pm‘rw_addr, pm‘states, access_count)
{2}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      access_count(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            write_rnd_dev(a, bl, cp) ##
                                             (λ (bl: list[Byte]): re-
      set_count(a, bl, cp))))
```

Hiding formulas: 2,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.3.2`.

`random_side_effects_side_effect_content_unchanged_write.1.3.3:`

```
{-1}  size(uidt(dt_uint)) > 0
------------------------------------------------------------------------
{1}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      write_rnd_dev(a, bl, cp) ##
                                       (λ (bl: list[Byte]): reset_count(a, bl, cp)))
{2}   side_effect_content_unchanged(pm‘rw_addr, pm‘states,
                                    λ (a: Address, bl: list[Byte], cp: bool):
                                      access_count(a, bl, cp) ##
                                       (λ (bl: list[Byte]):
                                            write_rnd_dev(a, bl, cp) ##
                                             (λ (bl: list[Byte]): re-
      set_count(a, bl, cp))))
```

Hiding formulas: 2,

Rewriting using side_effect_content_unchanged_composition, matching in *,

we get 3 subgoals:

`random_side_effects_side_effect_content_unchanged_write.1.3.3.1`:

{-1}	size(uidt(dt_uint)) > 0
{1}	transformer_invariant?(pm‘states, se_super_transformers(pm‘rw_addr, write_rnd_dev))
{2}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, λ (a: Address, bl: list[Byte], cp: bool): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): reset_count(a, bl, cp)))

Rewriting using transformer_invariant_all_transformers, matching in *,
Expanding the definition of se_super_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.3.3.1`.
`random_side_effects_side_effect_content_unchanged_write.1.3.3.2`:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, write_rnd_dev)
{2}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, λ (a: Address, bl: list[Byte], cp: bool): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): reset_count(a, bl, cp)))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.3.3.2`.
`random_side_effects_side_effect_content_unchanged_write.1.3.3.3`:

{-1}	size(uidt(dt_uint)) > 0
{1}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, reset_count)
{2}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, λ (a: Address, bl: list[Byte], cp: bool): write_rnd_dev(a, bl, cp) ## (λ (bl: list[Byte]): reset_count(a, bl, cp)))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_write.1.3.3.3`.
`random_side_effects_side_effect_content_unchanged_write.2`:

{1}	size(uidt(dt_uint)) > 0
{2}	side_effect_content_unchanged(pm‘rw_addr, pm‘states, rnd_dev_write_side_effect)

Using lemma dt_uint_size,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `random_side_effects_side_effect_content_unchanged_write.2`.
Q.E.D.

C.23.8 Challenge_Random_Device.random_device_plain_memory

Terse proof for `random_device_plain_memory`.

`random_device_plain_memory`:

{1} plain_memory?(pm_phy) ⊃ plain_memory?(pm)

Using lemma device_memory_plain_memory,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of is_device_plain_memory?,
Keeping (1) and hiding *,
Rewriting using random_side_effects_transformers_ok, matching in *,
Rewriting using random_side_effects_unchanged_invariant, matching in *,
Rewriting using random_side_effects_side_effect_content_unchanged_read, matching in *,
Rewriting using random_side_effects_side_effect_content_unchanged_write, matching in *,
Expanding the definition of pm,
Expanding the definition of random_device_pm,
Installing automatic rewrites from: subset_reflexive difference_subset
Expanding the definition of random_device_pm_mem,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `random_device_plain_memory`.
Q.E.D.

C.23.9 Challenge_Random_Device.unaligned_fails_write

Terse proof for `unaligned_fails_write`.

`unaligned_fails_write`:

{1} ∀ (*s*: Random_device_memory, *a*: Address, bl: (cons?[Byte])):
¬ disjoint?(address_block(*a*, length(bl)), address_block(base, size)) ∧
¬ (length(bl) = size(uidt(dt_uint)) ∧
(*a* = rnd_seed ∨ *a* = mem_cnt ∨ *a* = rnd_val))
∧
OK?(memory_write_side_effect(pm_phy‘mem)(*a*, bl, FALSE)(*s*‘state)) ∧
data(memory_write_side_effect(pm_phy‘mem)(*a*, bl, FALSE)(*s*‘state)) = bl
⊃ Fatal?(memory_write_list(pm‘mem)(*a*, bl)(*s*))

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_list,
Expanding the definition of pm,
Expanding the definition of random_device_pm,
Expanding the definition of random_device_pm_mem,
Expanding the definition of device_pm,
Expanding the definition of ##,
Expanding the definition of em_lift,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of rnd_dev_write_side_effect,
Expanding the definition of unaligned_access,
Expanding the definition of fatal_result,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unaligned_fails_write`.
Q.E.D.

C.23.10 Challenge_Random_Device.unaligned_fails_read

Terse proof for `unaligned_fails_read`.

`unaligned_fails_read`:

```
{1}  ∀ (s: Random_device_memory, a: Address, bl: (cons?[Byte])):
       ¬ disjoint?(address_block(a, length(bl)), address_block(base, size)) ∧
        ¬ (length(bl) = size(uidt(dt_uint)) ∧
             (a = rnd_seed ∨ a = mem_cnt ∨ a = rnd_val))
         ∧
         OK?(memory_read_list_nse(pm‘mem)(a, length(bl))(s)) ∧
          (LET ures =
                 (memory_read_list_nse(pm‘mem)(a, length(bl)) ##
                    (λ (bl1: list[Byte]):
                       em_lift
                         [Physical_memory, Random_device_state[Physical_memory, pm_phy],
                          list[Byte]]
                         (memory_read_side_effect(pm_phy‘mem)(a, bl1, FALSE))))
                   (s)
           IN OK?(ures) ∧ length(data(ures)) = length(bl))
        ⊃ Fatal?(memory_read_list(pm‘mem)(a, length(bl))(s))
```

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of pm,

Expanding the definition of random_device_pm,

Expanding the definition of random_device_pm_mem,

Expanding the definition of device_pm,

Expanding the definition of rnd_dev_read_side_effect,

Expanding the definition of unaligned_access,

Expanding the definition of ##,

Applying disjunctive simplification to flatten sequent,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on cons?(data(em_lift [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]] (memory_read_side_effect(pm_phy‘mem) (a!1, data(memory_read_list_nse(pm‘mem) (a!1, length(bl!1)) (s!1)), FALSE)) (state(memory_read_list_nse(pm‘mem) (a!1, length(bl!1)) (s!1))))),

we get 2 subgoals:

```
unaligned_fails_read.1:

{-1}  cons?(data(em_lift
                 [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[B
                 (memory_read_side_effect(pm_phy`mem)
                        (a′, data(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′)),
                          FALSE))
                 (state(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′)))))
{-2}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                 rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-3}  cons?[Byte](bl′)
{-4}  OK?(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′))
{-5}  OK?(em_lift[Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]
              (memory_read_side_effect(pm_phy`mem)
                     (a′, data(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′)),
                       FALSE))
              (state(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′))))
{-6}  length(data(em_lift
                  [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[
                  (memory_read_side_effect(pm_phy`mem)
                         (a′,
                           data(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′)),
                           FALSE))
                  (state(memory_read_list_nse(pm`mem)(a′, length(bl′))(s′)))))
       = length(bl′)
-------------------------------------------------------------------------------
{1}   disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{2}   (length(bl′) = size(uidt(dt_uint)) ∧
         (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
{3}   Fatal?(CASES IF null?(data(em_lift
                                  [Physical_memory,
                                    Random_device_state[Physical_memory, pm_phy], list
                                  (memory_read_side_effect(pm_phy`mem)
                                         (a′,
                                           data(memory_read_list_nse(pm`mem)
                                                                     (a′, length(bl′))
                                                                     (s′)),
                                           FALSE))
                                  (state(memory_read_list_nse(pm`mem)
                                                                    (a′, length(bl′))
                                                                    (s′)))))
                              ∨
                              length(data(em_lift
                                           [Physical_memory,
                                             Random_device_state[Physical_memory, pm_phy], l
                                           (memory_read_side_effect(pm_phy`mem)
                                                  (a′,
                                                    data(memory_read_list_nse(pm`mem)
                                                                              (a′, length(bl′)
                                                                              (s′)),
                                                    FALSE))
                                           (state(memory_read_list_nse(pm`mem)
                                                                             (a′, length(bl′))
                                                                             (s′)))))
                                = size(uidt(dt_uint))
                                ∧ (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val)
                             THEN ok_result(data(em_lift
                                                  [Physical_memory,
                                                    Random_device_state[Physical_memory, pm_
                                                    list[Byte]]
                                                  (memory_read_side_effect(pm_phy`mem)
                                                         (a′,
                                                           data(memory_read_list_nse(pm`mem)
```

Expanding the definition of fatal_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unaligned_fails_read.1`.

```
unaligned_fails_read.2:

{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
        (bl′)
{-2}  cons?[Byte](bl′)
{-3}  OK?(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′))
{-4}  OK?(em_lift[Physical_memory, Random_device_state[Physical_memory, pm_phy], list[Byte]]
            (memory_read_side_effect(pm_phy‘mem)
                   (a′, data(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′)),
                     FALSE))
            (state(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′))))
{-5}  length(data(em_lift
                    [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[
                    (memory_read_side_effect(pm_phy‘mem)
                           (a′,
                             data(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′)),
                             FALSE))
                    (state(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′)))))
        = length(bl′)
-------------------------------------------------------------------------------
{1}   cons?(data(em_lift
                   [Physical_memory, Random_device_state[Physical_memory, pm_phy], list[P
                   (memory_read_side_effect(pm_phy‘mem)
                          (a′, data(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′)),
                            FALSE))
                   (state(memory_read_list_nse(pm‘mem)(a′, length(bl′))(s′)))))
{2}   disjoint?(address_block(a′, length(bl′)), address_block(base, size))
{3}   (length(bl′) = size(uidt(dt_uint)) ∧
          (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val))
{4}   Fatal?(CASES IF null?(data(em_lift
                                      [Physical_memory,
                                        Random_device_state[Physical_memory, pm_phy], list
                                      (memory_read_side_effect(pm_phy‘mem)
                                             (a′,
                                               data(memory_read_list_nse(pm‘mem)
                                                                         (a′, length(bl′))
                                                                         (s′)),
                                                 FALSE))
                                      (state(memory_read_list_nse(pm‘mem)
                                                                  (a′, length(bl′))
                                                                  (s′)))))
                                ∨
                                length(data(em_lift
                                              [Physical_memory,
                                                Random_device_state[Physical_memory, pm_phy], l
                                              (memory_read_side_effect(pm_phy‘mem)
                                                     (a′,
                                                       data(memory_read_list_nse(pm‘mem)
                                                                                 (a′, length(bl′)
                                                                                 (s′)),
                                                         FALSE))
                                              (state(memory_read_list_nse(pm‘mem)
                                                                          (a′, length(bl′))
                                                                          (s′)))))
                                  = size(uidt(dt_uint))
                                  ∧ (a′ = rnd_seed ∨ a′ = mem_cnt ∨ a′ = rnd_val)
                                THEN ok_result(data(em_lift
                                                      [Physical_memory,
                                                        Random_device_state[Physical_memory, pm_
                                                        list[Byte]]
                                                      (memory_read_side_effect(pm_phy‘mem)
                                                             (a′,
                                                               data(memory_read_list_nse(pm‘mem)
```

Keeping (-2 -5 1) and hiding *,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unaligned_fails_read.2`.
Q.E.D.

C.24 Proofs for CommaExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.25 Proofs for Composition_Lifted_Statement_Lifted_Expression (state-transformer.pvs)

This theory contains no provable formal statements.

C.26 Proofs for Composition_Lifted_Statement_Lifted_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.27 Proofs for Composition_Lifted_Statement_Simple_Expression (state-transformer.pvs)

This theory contains no provable formal statements.

C.28 Proofs for Composition_Lifted_Statement_Simple_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.29 Proofs for Composition_Simple_Expression_Data_Expression (state-transformer.pvs)

C.29.1 Composition_Simple_Expression_Data_Expression.super_comp_expr_expr_TCC1

Terse proof for `super_comp_expr_expr_TCC1`.
`super_comp_expr_expr_TCC1:`

```
{1}  ∀ (expr: [State → ExprResult[State, Data1]], s, state: State):
       expr_2_super(expr)(s) = OK(state) ⊃
         OK?[State](expr_2_super[State, Data1](expr)(s)) ∨
           abnormal?[State](expr_2_super[State, Data1](expr)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_comp_expr_expr_TCC1`.
Q.E.D.

C.29.2 Composition_Simple_Expression_Data_Expression.super_comp_expr_expr_TCC2

Terse proof for `super_comp_expr_expr_TCC2`.
`super_comp_expr_expr_TCC2:`

```
{1}  ∀ (expr: [State → ExprResult[State, Data1]], s, state: State):
       expr_2_super(expr)(s) = OK(state) ⊃ OK?[State, Data1](expr(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_comp_expr_expr_TCC2`.
Q.E.D.

C.29.3 Composition_Simple_Expression_Data_Expression.super_comp_expr_expr

Terse proof for `super_comp_expr_expr`.
`super_comp_expr_expr:`

```
{1}  ∀ (expr: [State → ExprResult[State, Data1]],
          fexpr: [Data1 → [State → ExprResult[State, Data2]]], s: State):
       expr_2_super(expr ## fexpr)(s) =
        CASES expr_2_super(expr)(s) OF
          OK(state):
            expr_2_super(fexpr(data(expr(s))))(state(expr_2_super(expr)(s)))
          ELSE expr_2_super(expr)(s)
          ENDCASES
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_comp_expr_expr`.

Q.E.D.

C.30 Proofs for Composition_Simple_Expression_Data_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.31 Proofs for Composition_Simple_Expression_Simple_Expression (state-transformer.pvs)

C.31.1 Composition_Simple_Expression_Simple_Expression.comp_expr_forget_expr

Terse proof for `comp_expr_forget_expr`.

`comp_expr_forget_expr`:

```
{1}  ∀ (expr1: [State → ExprResult[State, Data1]],
          expr2: [State → ExprResult[State, Data2]], s: State):
       (expr1 ## expr2)(s) =
        CASES expr1(s) OF
          OK(state, data): expr2(state),
          Exception(ex_type, state): Exception(ex_type, state),
          Fatal: Fatal,
          Hang: Hang
          ENDCASES
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `comp_expr_forget_expr`.

Q.E.D.

C.31.2 Composition_Simple_Expression_Simple_Expression.super_comp_expr_forget_expr

Terse proof for `super_comp_expr_forget_expr`.

`super_comp_expr_forget_expr`:

```
{1}  ∀ (expr1: [State → ExprResult[State, Data1]],
          expr2: [State → ExprResult[State, Data2]], s: State):
       expr_2_super(expr1 ## expr2)(s) =
        CASES expr_2_super(expr1)(s) OF OK(state): expr_2_super(expr2)(state)
          ELSE expr_2_super(expr1)(s)
          ENDCASES
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `super_comp_expr_forget_expr`.
Q.E.D.

C.31.3 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_TC

Terse proof for `comp_simple_expr_simple_expr_TCC1`.
`comp_simple_expr_simple_expr_TCC1:`

```
{1}  ∀ (expr1: [State → ExprResult[State, Data1]], s: State):
       OK?(expr1(s)) ⊃
        OK?[State, Data1](expr1(s)) ∨ Exception?[State, Data1](expr1(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_TCC1`.
Q.E.D.

C.31.4 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr

Terse proof for `comp_simple_expr_simple_expr`.
`comp_simple_expr_simple_expr:`

```
{1}  ∀ (expr1: [State → ExprResult[State, Data1]],
           expr2: [State → ExprResult[State, Data2]], s: State):
       OK?(expr1(s)) ⊃ expr2(state(expr1(s))) = (expr1 ## expr2)(s)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr`.
Q.E.D.

C.31.5 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_ok

Terse proof for `comp_simple_expr_simple_expr_ok_TCC1`.
`comp_simple_expr_simple_expr_ok_TCC1:`

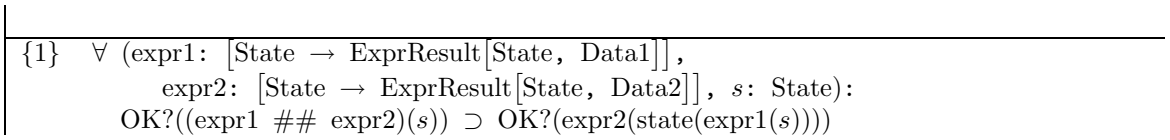
```
{1}  ∀ (expr1: [State → ExprResult[State, Data1]],
           expr2: [State → ExprResult[State, Data2]], s: State):
       OK?((expr1 ## expr2)(s)) ⊃
        OK?[State, Data1](expr1(s)) ∨ Exception?[State, Data1](expr1(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_ok_TCC1`.
Q.E.D.

C.31.6 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_ok

Terse proof for `comp_simple_expr_simple_expr_ok`.

`comp_simple_expr_simple_expr_ok`:

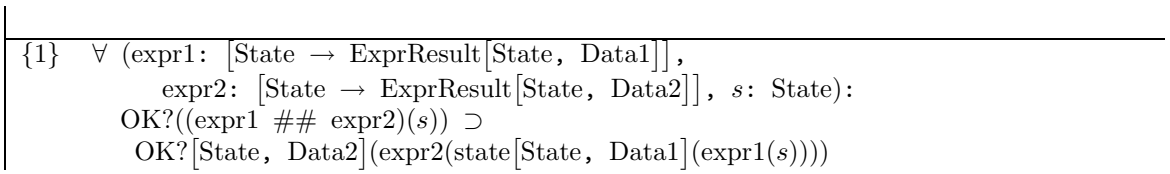


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_ok`.
Q.E.D.

C.31.7 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_data_TCC

Terse proof for `comp_simple_expr_simple_expr_data_TCC1`.

`comp_simple_expr_simple_expr_data_TCC1`:

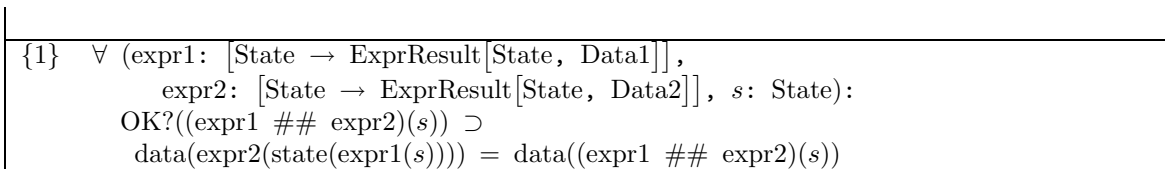


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_data_TCC1`.
Q.E.D.

C.31.8 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_data

Terse proof for `comp_simple_expr_simple_expr_data`.

`comp_simple_expr_simple_expr_data`:

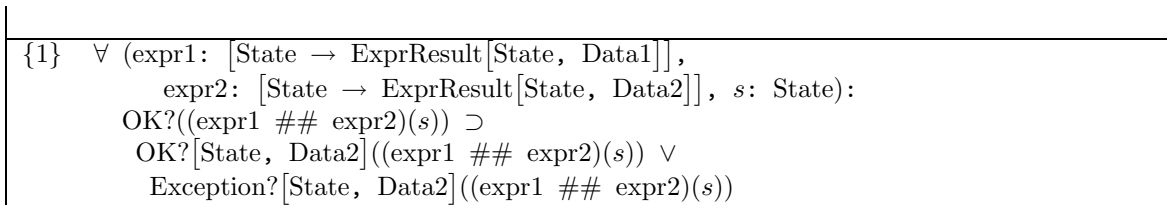


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_simple_expr_simple_expr_data`.
Q.E.D.

C.31.9 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_state_TCC

Terse proof for `comp_simple_expr_simple_expr_state_TCC1`.

comp_simple_expr_simple_expr_state_TCC1:



Trying repeated skolemization, instantiation, and if-lifting,

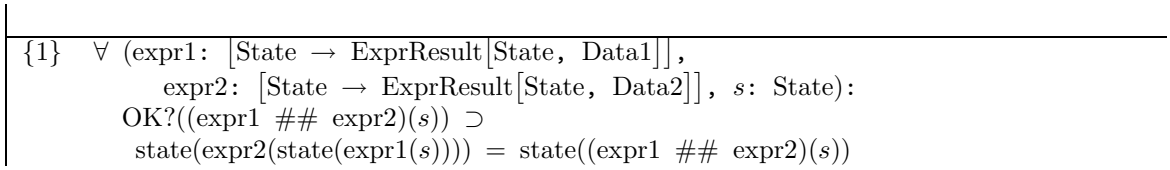
This completes the proof of comp_simple_expr_simple_expr_state_TCC1.

Q.E.D.

C.31.10 Composition_Simple_Expression_Simple_Expression.comp_simple_expr_simple_expr_sta

Terse proof for comp_simple_expr_simple_expr_state.

comp_simple_expr_simple_expr_state:



Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of comp_simple_expr_simple_expr_state.

Q.E.D.

C.32 Proofs for Composition_Simple_Statement_Lifted_Expression (state-transformer.pvs)

This theory contains no provable formal statements.

C.33 Proofs for Composition_Simple_Statement_Lifted_Statement (state-transformer.pvs)

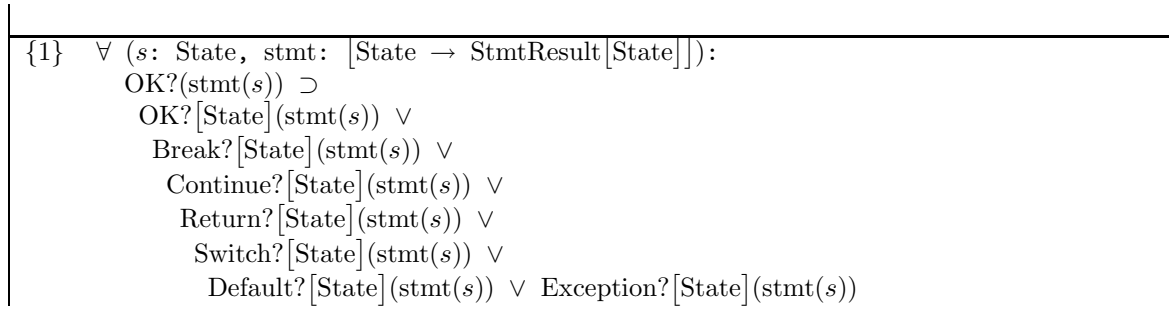
This theory contains no provable formal statements.

C.34 Proofs for Composition_Simple_Statement_Simple_Expression (state-transformer.pvs)

C.34.1 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_TCC1

Terse proof for `comp_simple_stmt_simple_expr_TCC1`.

`comp_simple_stmt_simple_expr_TCC1`:



Trying repeated skolemization, instantiation, and if-lifting,

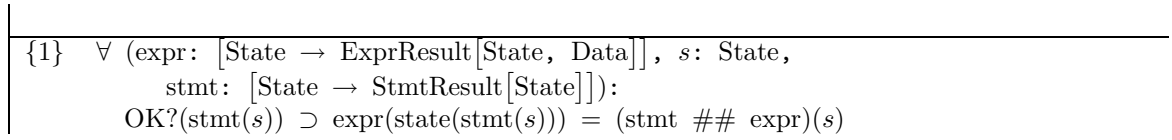
This completes the proof of `comp_simple_stmt_simple_expr_TCC1`.

Q.E.D.

C.34.2 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr

Terse proof for `comp_simple_stmt_simple_expr`.

`comp_simple_stmt_simple_expr`:



Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `comp_simple_stmt_simple_expr`.

Q.E.D.

C.34.3 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_ok_TCC1

Terse proof for `comp_simple_stmt_simple_expr_ok_TCC1`.

comp_simple_stmt_simple_expr_ok_TCC1:

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃
OK?[State](stmt(s)) ∨
Break?[State](stmt(s)) ∨
Continue?[State](stmt(s)) ∨
Return?[State](stmt(s)) ∨
Switch?[State](stmt(s)) ∨
Default?[State](stmt(s)) ∨ Exception?[State](stmt(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_ok_TCC1.
Q.E.D.

C.34.4 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_ok

Terse proof for comp_simple_stmt_simple_expr_ok.

comp_simple_stmt_simple_expr_ok:

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃ OK?(expr(state(stmt(s))))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_ok.
Q.E.D.

C.34.5 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_da

Terse proof for comp_simple_stmt_simple_expr_data_TCC1.

comp_simple_stmt_simple_expr_data_TCC1:

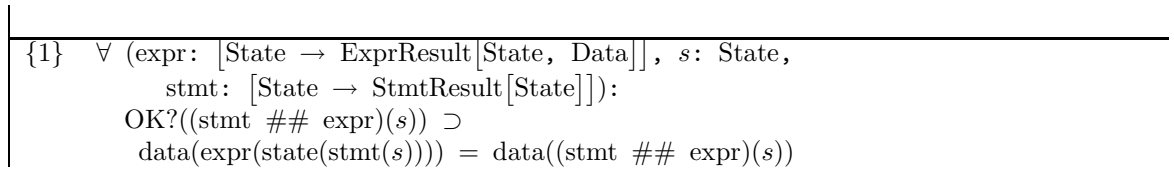
{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃ OK?[State, Data](expr(state[State](stmt(s))))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_data_TCC1.
Q.E.D.

C.34.6 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_da

Terse proof for comp_simple_stmt_simple_expr_data.

comp_simple_stmt_simple_expr_data:

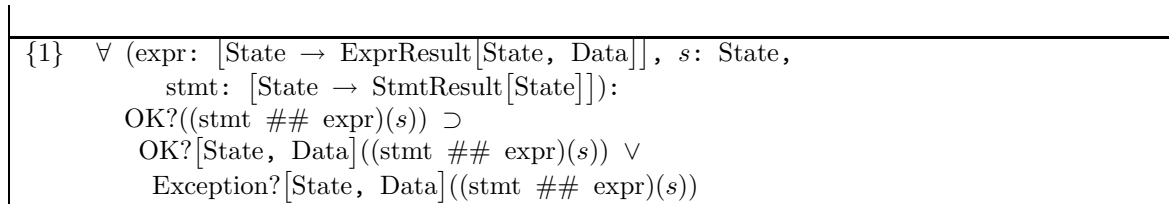


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_data.
Q.E.D.

C.34.7 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_state_TC

Terse proof for comp_simple_stmt_simple_expr_state_TCC1.

comp_simple_stmt_simple_expr_state_TCC1:

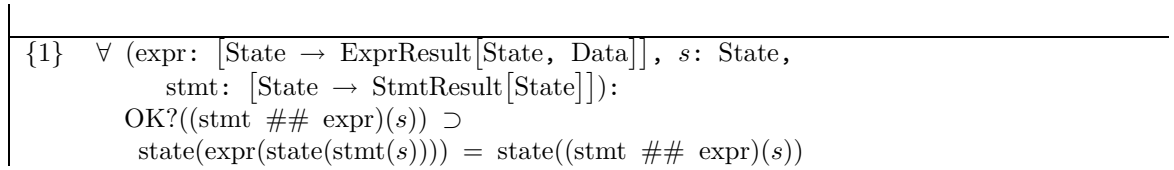


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_state_TCC1.
Q.E.D.

C.34.8 Composition_Simple_Statement_Simple_Expression.comp_simple_stmt_simple_expr_state

Terse proof for comp_simple_stmt_simple_expr_state.

comp_simple_stmt_simple_expr_state:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_simple_stmt_simple_expr_state.
Q.E.D.

C.35 Proofs for Composition_Simple_Statement_Simple_Statement (state-transformer.pvs)

This theory contains no provable formal statements.

C.36 Proofs for ConditionalExpression (expressions.pvs)

This theory contains no provable formal statements.

C.37 Proofs for Control_Register_Datatype (paging-data.pvs)

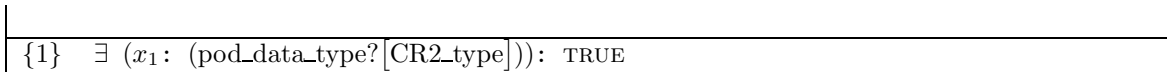
C.37.1 Control_Register_Datatype.cr0_data_type_TCC1

Terse proof for `cr0_data_type_TCC1`.
`cr0_data_type_TCC1:`

Rewriting using cr0_data_type_exists, matching in *,
This completes the proof of `cr0_data_type_TCC1`.
Q.E.D.

C.37.2 Control_Register_Datatype.cr2_data_type_TCC1

Terse proof for `cr2_data_type_TCC1`.
`cr2_data_type_TCC1:`



Rewriting using cr2_data_type_exists, matching in *,
This completes the proof of `cr2_data_type_TCC1`.
Q.E.D.

C.37.3 Control_Register_Datatype.cr4_data_type_TCC1

Terse proof for `cr4_data_type_TCC1`.
`cr4_data_type_TCC1:`

Rewriting using cr4_data_type_exists, matching in *,
This completes the proof of `cr4_data_type_TCC1`.
Q.E.D.

C.38 Proofs for Control_Register_Types (paging-data.pvs)

This theory contains no provable formal statements.

C.39 Proofs for Conversions (conversions.pvs)

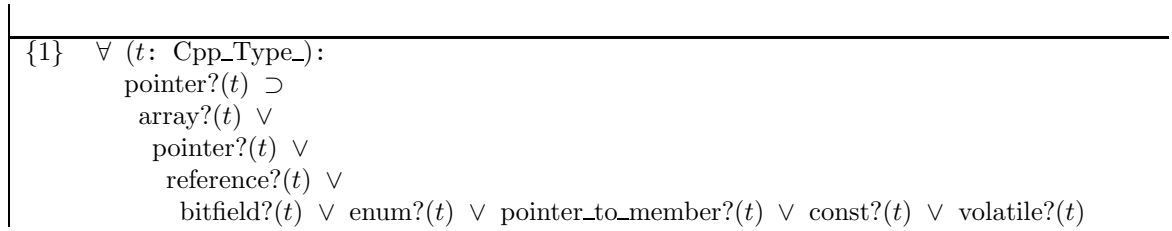
This theory contains no provable formal statements.

C.40 Proofs for Cpp_Deep_Type_Wellformedness (types.pvs)

C.40.1 Cpp_Deep_Type_Wellformedness.no_pointers_to_bitfield?_TCC1

Terse proof for `no_pointers_to_bitfield?_TCC1`.

`no_pointers_to_bitfield?_TCC1:`



Trying repeated skolemization, instantiation, and if-lifting,

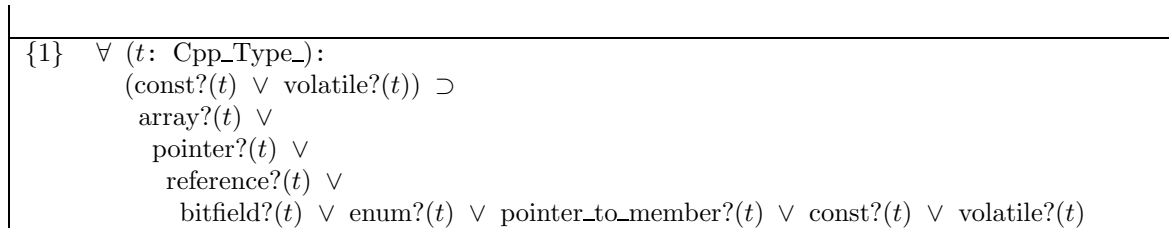
This completes the proof of `no_pointers_to_bitfield?_TCC1`.

Q.E.D.

C.40.2 Cpp_Deep_Type_Wellformedness.no_cv_references?_TCC1

Terse proof for `no_cv_references?_TCC1`.

`no_cv_references?_TCC1:`



Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `no_cv_references?_TCC1`.

Q.E.D.

C.40.3 Cpp_Deep_Type_Wellformedness.no_reference_to_reference?_TCC1

Terse proof for `no_reference_to_reference?_TCC1`.

no_reference_to_reference?_TCC1:

{1} ∀ (t: Cpp_Type_):
reference?(t) ⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_reference_to_reference?_TCC1.
Q.E.D.

C.40.4 Cpp_Deep_Type_Wellformedness.no_array_of_references?_TCC1

Terse proof for no_array_of_references?_TCC1.

no_array_of_references?_TCC1:

{1} ∀ (t: Cpp_Type_):
array?(t) ⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_array_of_references?_TCC1.
Q.E.D.

C.40.5 Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_para

Terse proof for no_pointer_or_ref_to_incomplete_array_parameter?_TCC1.

no_pointer_or_ref_to_incomplete_array_parameter?_TCC1:

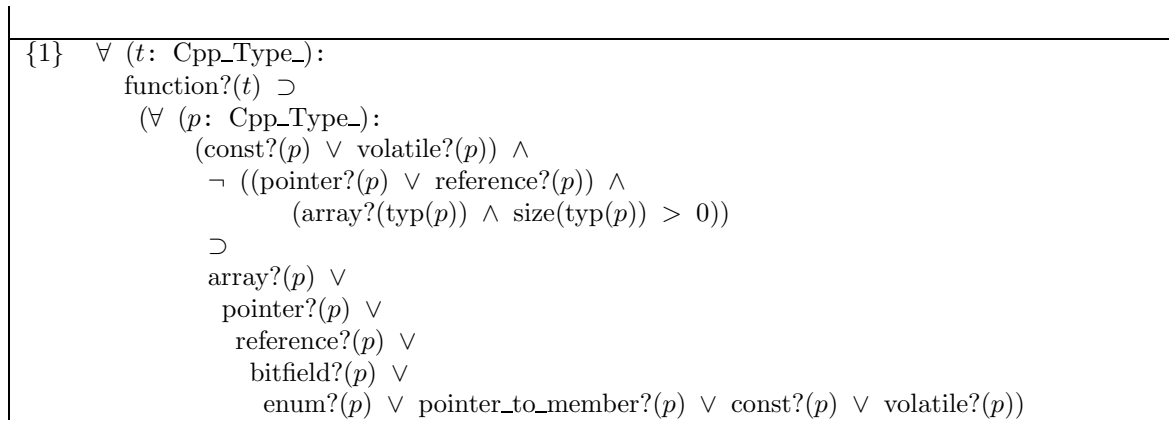
{1} ∀ (t: Cpp_Type_):
function?(t) ⊃
(∀ (p: Cpp_Type_):
(pointer?(p) ∨ reference?(p)) ⊃
array?(p) ∨
pointer?(p) ∨
reference?(p) ∨
bitfield?(p) ∨
enum?(p) ∨ pointer_to_member?(p) ∨ const?(p) ∨ volatile?(p))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of no_pointer_or_ref_to_incomplete_array_parameter?_TCC1.
Q.E.D.

C.40.6 Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?_

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC2`.

`no_pointer_or_ref_to_incomplete_array_parameter?_TCC2`:



Trying repeated skolemization, instantiation, and if-lifting,

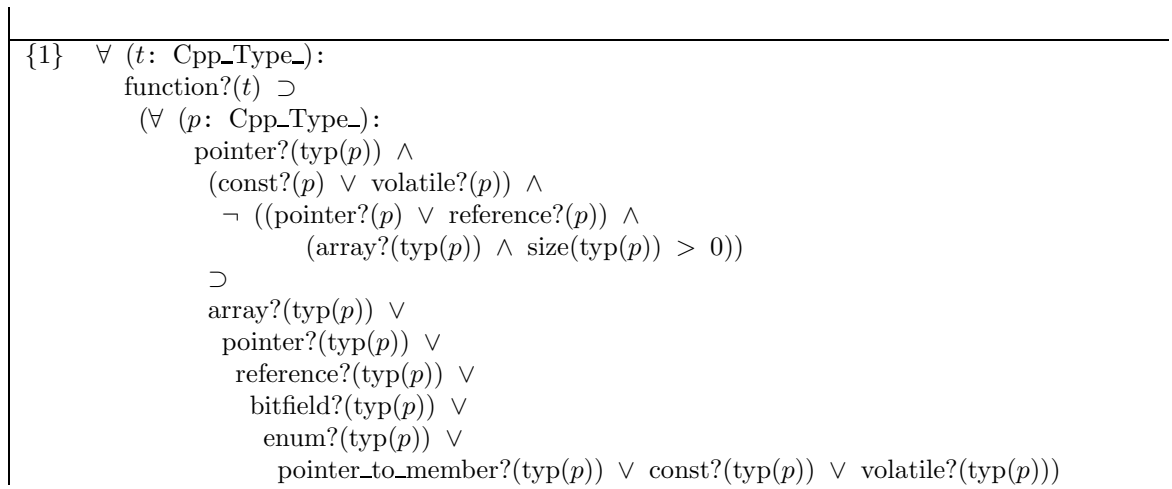
This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC2`.

Q.E.D.

C.40.7 Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?_

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC3`.

`no_pointer_or_ref_to_incomplete_array_parameter?_TCC3`:



Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC3`.

Q.E.D.

C.40.8 Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_para

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC4`.

`no_pointer_or_ref_to_incomplete_array_parameter?_TCC4`:

```
{1}   ∀ (t: Cpp_Type_):
        function?(t) ⊃
         (∀ (p: Cpp_Type_):
            const?(p) ∧
             ¬ ((const?(p) ∨ volatile?(p)) ∧
                   pointer?(typ(p)) ∧
                    (array?(typ(typ(p))) ∧ size(typ(typ(p))) > 0))
              ∧
             ¬ ((pointer?(p) ∨ reference?(p)) ∧
                   (array?(typ(p)) ∧ size(typ(p)) > 0))
            ⊃
            array?(p) ∨
             pointer?(p) ∨
              reference?(p) ∨
               bitfield?(p) ∨
                enum?(p) ∨ pointer_to_member?(p) ∨ const?(p) ∨ volatile?(p))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC4`.

Q.E.D.

C.40.9 Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_para

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC5`.

`no_pointer_or_ref_to_incomplete_array_parameter?_TCC5`:

```
{1}   ∀ (t: Cpp_Type_):
        function?(t) ⊃
         (∀ (p: Cpp_Type_):
            volatile?(typ(p)) ∧
             const?(p) ∧
              ¬ ((const?(p) ∨ volatile?(p)) ∧
                    pointer?(typ(p)) ∧
                     (array?(typ(typ(p))) ∧ size(typ(typ(p))) > 0))
               ∧
              ¬ ((pointer?(p) ∨ reference?(p)) ∧
                    (array?(typ(p)) ∧ size(typ(p)) > 0))
             ⊃
             array?(typ(p)) ∨
              pointer?(typ(p)) ∨
               reference?(typ(p)) ∨
                bitfield?(typ(p)) ∨
                 enum?(typ(p)) ∨
                  pointer_to_member?(typ(p)) ∨ const?(typ(p)) ∨ volatile?(typ(p)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC5`.
Q.E.D.

C.40.10 Cpp_Deep_Type_Wellformedness.no_pointer_or_ref_to_incomplete_array_parameter?_

Terse proof for `no_pointer_or_ref_to_incomplete_array_parameter?_TCC6`.
`no_pointer_or_ref_to_incomplete_array_parameter?_TCC6:`

```
{1}  ∀ (t: Cpp_Type_):
       function?(t) ⊃
        (∀ (p: Cpp_Type_):
           pointer?(typ(typ(p))) ∧
            volatile?(typ(p)) ∧
             const?(p) ∧
              ¬ ((const?(p) ∨ volatile?(p)) ∧
                    pointer?(typ(p)) ∧
                     (array?(typ(typ(p))) ∧ size(typ(typ(p))) > 0))
               ∧
               ¬ ((pointer?(p) ∨ reference?(p)) ∧
                     (array?(typ(p)) ∧ size(typ(p)) > 0))
           ⊃
           array?(typ(typ(p))) ∨
            pointer?(typ(typ(p))) ∨
             reference?(typ(typ(p))) ∨
              bitfield?(typ(typ(p))) ∨
               enum?(typ(typ(p))) ∨
                pointer_to_member?(typ(typ(p))) ∨
                 const?(typ(typ(p))) ∨ volatile?(typ(typ(p))))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `no_pointer_or_ref_to_incomplete_array_parameter?_TCC6`.
Q.E.D.

C.40.11 Cpp_Deep_Type_Wellformedness.bitfield_underlying_integral_or_enum_type?_TCC1

Terse proof for `bitfield_underlying_integral_or_enum_type?_TCC1`.
`bitfield_underlying_integral_or_enum_type?_TCC1:`

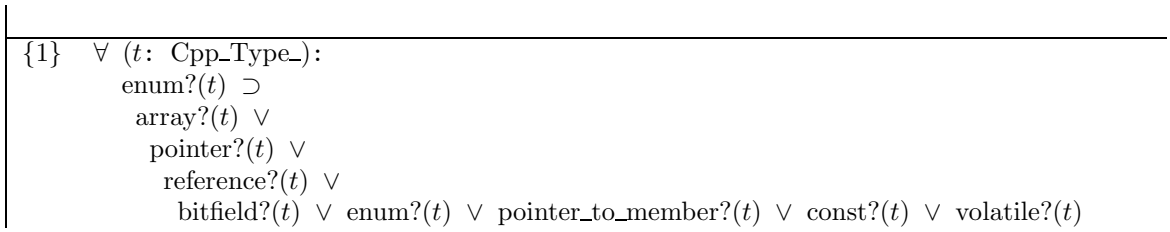
```
{1}  ∀ (t: Cpp_Type_):
       bitfield?(t) ⊃
        array?(t) ∨
         pointer?(t) ∨
          reference?(t) ∨
           bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `bitfield_underlying_integral_or_enum_type?_TCC1`.
Q.E.D.

C.40.12 Cpp_Deep_Type_Wellformedness.enum_underlying_integral?_TCC1

Terse proof for `enum_underlying_integral?_TCC1`.

`enum_underlying_integral?_TCC1:`

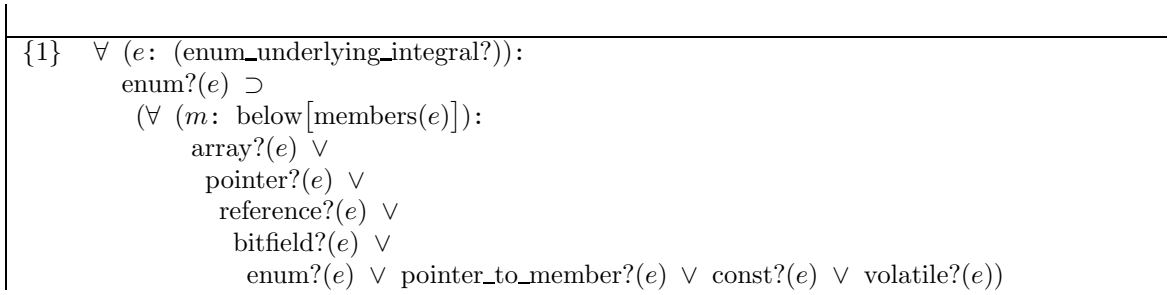


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `enum_underlying_integral?_TCC1`.
Q.E.D.

C.40.13 Cpp_Deep_Type_Wellformedness.enum_constants?_TCC1

Terse proof for `enum_constants?_TCC1`.

`enum_constants?_TCC1:`

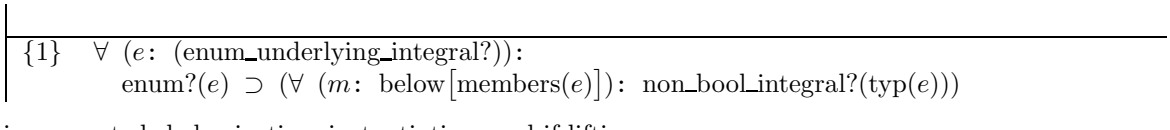


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `enum_constants?_TCC1`.
Q.E.D.

C.40.14 Cpp_Deep_Type_Wellformedness.enum_constants?_TCC2

Terse proof for `enum_constants?_TCC2`.

`enum_constants?_TCC2:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `enum_constants?_TCC2`.
Q.E.D.

C.40.15 Cpp_Deep_Type_Wellformedness.const_volatile?_TCC1

Terse proof for `const_volatile?_TCC1`.

`const_volatile?_TCC1:`

{1} ∀ (t: Cpp_Type_):
volatile?(t) ⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `const_volatile?_TCC1`.
Q.E.D.

C.40.16 Cpp_Deep_Type_Wellformedness.const_stutter?_TCC1

Terse proof for `const_stutter?_TCC1`.

`const_stutter?_TCC1:`

{1} ∀ (t: Cpp_Type_):
const?(t) ⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `const_stutter?_TCC1`.
Q.E.D.

C.40.17 Cpp_Deep_Type_Wellformedness.cv_array?_TCC1

Terse proof for `cv_array?_TCC1`.

`cv_array?_TCC1:`

{1} ∀ (t: Cpp_Type_):
(volatile?(t) ∨ const?(t)) ⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cv_array?_TCC1`.
Q.E.D.

C.40.18 Cpp_Deep_Type_Wellformedness.cv_base_TCC1

Terse proof for `cv_base_TCC1`.

cv_base_TCC1:

{1} well_founded?(λ (x: Cpp_Type, y: Cpp_Type):
restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean]
(≪)(x, y))

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Type],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of cv_base_TCC1.
Q.E.D.

C.40.19 Cpp_Deep_Type_Wellformedness.cv_base_TCC2

Terse proof for cv_base_TCC2.
cv_base_TCC2:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_): typ = const(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of cv_base_TCC2.
Q.E.D.

C.40.20 Cpp_Deep_Type_Wellformedness.cv_base_TCC3

Terse proof for cv_base_TCC3.
cv_base_TCC3:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃
restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean](≪)(t, typ)

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of ■,
which is trivially true.
This completes the proof of cv_base_TCC3.
Q.E.D.

C.40.21 Cpp_Deep_Type_Wellformedness.cv_base_TCC4

Terse proof for cv_base_TCC4.
cv_base_TCC4:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_): typ = volatile(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_TCC4`.
Q.E.D.

C.40.22 Cpp_Deep_Type_Wellformedness.cv_base_TCC5

Terse proof for `cv_base_TCC5`.

`cv_base_TCC5:`

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
       typ = volatile(t) ⊃
        restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean](≪)(t, typ)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of restrict,
Replacing using formula -2,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `cv_base_TCC5`.
Q.E.D.

C.40.23 Cpp_Deep_Type_Wellformedness.subterm_cv_base

Terse proof for `subterm_cv_base`.

`subterm_cv_base:`

```
{1}  ∀ (typ: Cpp_Type): subterm(cv_base(typ), typ)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of cv_base,
Expanding the definition of cv_base,
Lifting IF-conditions to the top level,
Expanding the definition of subterm,
Expanding the definition of subterm,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of Cpp_Type?,
Expanding the definition of const_volatile?,
Expanding the definition of const_stutter?,
Expanding the definition of volatile_stutter?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 5 subgoals:

`subterm_cv_base.1:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ′) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
(volatile?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(const?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(volatile?(t) $\supset$ $\neg$ volatile?(typ(t))) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)
{-2} const?(typ′)
{-3} const?(typ(typ′))

{1} subterm(cv_base(typ(typ(typ′))), typ′)

Instantiating the top quantifier in -1 with the terms: (typ!1),

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `subterm_cv_base.1`.

`subterm_cv_base.2:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
(volatile?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(const?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(volatile?(t) $\supset$ $\neg$ volatile?(typ(t))) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ$'$)

{-3} volatile?(typ(typ$'$))

{1} subterm(cv_base(typ(typ(typ$'$))), typ$'$)

Expanding the definition of cv_base,

Instantiating the top quantifier in -1 with the terms: (typ(typ!1)),

Expanding the definition of subterm,

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `subterm_cv_base.2`.

`subterm_cv_base.3:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ′) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
(volatile?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(const?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(volatile?(t) $\supset$ $\neg$ volatile?(typ(t))) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{1} const?(typ′)
{2} volatile?(typ′)
{3} subterm(typ′, typ′)

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `subterm_cv_base.3`.

`subterm_cv_base.4:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ′) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
(volatile?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(const?(t) $\supset$ $\neg$ const?(typ(t))) $\wedge$
(volatile?(t) $\supset$ $\neg$ volatile?(typ(t))) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ(typ′))

{-3} volatile?(typ′)

{1} subterm(cv_base(typ(typ(typ′))), typ′)

Instantiating the top quantifier in -1 with the terms: (typ!1),

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `subterm_cv_base.4`.

`subterm_cv_base.5:`

{-1} ∀ (t: Cpp_Type_):
subterm(t, typ′) ⊃
no_pointers_to_bitfield?(t) ∧
no_pointers_to_references?(t) ∧
no_cv_references?(t) ∧
no_reference_to_reference?(t) ∧
no_reference_to_bitfields?(t) ∧
no_pointer_to_member_to_reference?(t) ∧
no_pointer_to_member_to_cv_void?(t) ∧
no_cv_void_parameter?(t) ∧
no_array_of_references?(t) ∧
no_array_of_cv_void?(t) ∧
no_array_of_function?(t) ∧
no_array_of_abstract_class?(t) ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧
no_array_return_type?(t) ∧
no_function_return_type?(t) ∧
bitfield_underlying_integral_or_enum_type?(t) ∧
cv_array?(t) ∧
enum_underlying_integral?(t) ∧
enum_constants?(t) ∧
(volatile?(t) ⊃ ¬ const?(typ(t))) ∧
(const?(t) ⊃ ¬ const?(typ(t))) ∧
(volatile?(t) ⊃ ¬ volatile?(typ(t))) ∧
no_cv_class?(t) ∧
no_cv_union?(t) ∧
no_cv_function?(t) ∧
no_reference_to_void?(t) ∧
no_cv_void?(t) ∧ no_array_of_bitfields?(t)
{-2} volatile?(typ(typ′))
{-3} volatile?(typ′)

{1} subterm(cv_base(typ(typ(typ′))), typ′)

Instantiating the top quantifier in -1 with the terms: (typ!1),
Expanding the definition of subterm,
which is trivially true.
This completes the proof of `subterm_cv_base.5`.
Q.E.D.

C.40.24 Cpp_Deep_Type_Wellformedness.cv_base_result

Terse proof for `cv_base_result`.

`cv_base_result:`

{1} ∀ (P: {Q: PRED[Cpp_Type_] | (Q ⊆ interpreted?)}, typ: Cpp_Subtype(cv(P))):
P(cv_base(typ))

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,

Installing automatic rewrites from: cv_base
Case splitting on FORALL (typ: (P!1)): NOT const?(typ) AND NOT volatile?(typ),
we get 2 subgoals:
`cv_base_result.1:`

{-1} $\forall$ (typ: (P')): $\neg$ const?(typ) $\wedge$ $\neg$ volatile?(typ)
{-2} ($P' \subseteq$ interpreted?)
{-3} Cpp_Type?(typ$'$)
{-4} (P'(typ$'$) $\vee$ (const?(typ$'$) $\wedge$ P'(typ(typ$'$)))) $\vee$
(P'(typ$'$) $\vee$ (volatile?(typ$'$) $\wedge$ P'(typ(typ$'$)))) $\vee$
(const?(typ$'$) $\wedge$ volatile?(typ(typ$'$)) $\wedge$ P'(typ(typ(typ$'$))))

{1} P'(cv_base(typ$'$))

Hiding formulas: (-2 -3),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`cv_base_result.1.1:`

{-1} $\forall$ (typ: (P')): $\neg$ const?(typ) $\wedge$ $\neg$ volatile?(typ)
{-2} P'(typ$'$)

{1} P'(cv_base(typ$'$))

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_result.1.1`.
`cv_base_result.1.2:`

{-1} $\forall$ (typ: (P')): $\neg$ const?(typ) $\wedge$ $\neg$ volatile?(typ)
{-2} const?(typ$'$)
{-3} P'(typ(typ$'$))

{1} P'(cv_base(typ(typ$'$)))

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_result.1.2`.
`cv_base_result.1.3:`

{-1} $\forall$ (typ: (P')): $\neg$ const?(typ) $\wedge$ $\neg$ volatile?(typ)
{-2} const?(typ$'$)
{-3} volatile?(typ(typ$'$))
{-4} P'(typ(typ(typ$'$)))

{1} P'(cv_base(typ(typ(typ$'$))))

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_result.1.3`.
`cv_base_result.1.4:`

{-1} $\forall$ (typ: (P')): $\neg$ const?(typ) $\wedge$ $\neg$ volatile?(typ)
{-2} P'(typ(typ$'$))
{-3} volatile?(typ$'$)

{1} P'(cv_base(typ(typ$'$)))

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_result.1.4`.

`cv_base_result.2:`

{-1} $(P' \subseteq \text{interpreted?})$
{-2} $\text{Cpp\_Type?}(\text{typ}')$
{-3} $(P'(\text{typ}') \vee (\text{const?}(\text{typ}') \wedge P'(\text{typ}(\text{typ}')))) \vee$
$(P'(\text{typ}') \vee (\text{volatile?}(\text{typ}') \wedge P'(\text{typ}(\text{typ}')))) \vee$
$(\text{const?}(\text{typ}') \wedge \text{volatile?}(\text{typ}(\text{typ}')) \wedge P'(\text{typ}(\text{typ}(\text{typ}'))))$

{1} $\forall\ (\text{typ}: (P')): \neg\ \text{const?}(\text{typ}) \wedge \neg\ \text{volatile?}(\text{typ})$
{2} $P'(\text{cv\_base}(\text{typ}'))$

Keeping (-1 1) and hiding *,
Repeatedly Skolemizing and flattening,
Expanding the definition of subset?,
Expanding the definition of member,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cv_base_result.2`.
Q.E.D.

C.40.25 Cpp_Deep_Type_Wellformedness.cv_base_result2

Terse proof for `cv_base_result2`.

`cv_base_result2:`

{1} $\forall\ (\text{typ}: \text{Cpp\_Type}): \neg\ \text{const?}(\text{typ}) \wedge \neg\ \text{volatile?}(\text{typ}) \supset \text{cv\_base}(\text{typ}) = \text{typ}$

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cv_base_result2`.
Q.E.D.

C.40.26 Cpp_Deep_Type_Wellformedness.cv_base_not_const

Terse proof for `cv_base_not_const`.

`cv_base_not_const:`

{1} $\forall\ (\text{typ}: \text{Cpp\_Type}): \neg\ \text{const?}(\text{cv\_base}(\text{typ}))$

Repeatedly Skolemizing and flattening,
Expanding the definition of cv_base,
Expanding the definition of cv_base,
Expanding the definition of Cpp_Type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:

`cv_base_not_const.1:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ$'$)

{-3} const?(typ(typ$'$))

{-4} const?(cv_base(typ(typ(typ$'$))))

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of const_stutter?,

which is trivially true.

This completes the proof of `cv_base_not_const.1`.

`cv_base_not_const.2:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ′) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)
{-2} const?(typ′)
{-3} const?(cv_base(typ(typ(typ′))))
{-4} volatile?(typ(typ′))

Expanding the definition of cv_base,

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of subterm,

Expanding the definition of volatile_stutter?,

Expanding the definition of const_volatile?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_base_not_const.2`.

`cv_base_not_const.3:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ(typ$'$))

{-3} const?(cv_base(typ(typ(typ$'$))))

{-4} volatile?(typ$'$)

Instantiating quantified variables,

Expanding the definition of const_volatile?,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `cv_base_not_const.3`.

`cv_base_not_const.4:`

{-1} ∀ (t: Cpp_Type_):
subterm(t, typ′) ⊃
no_pointers_to_bitfield?(t) ∧
no_pointers_to_references?(t) ∧
no_cv_references?(t) ∧
no_reference_to_reference?(t) ∧
no_reference_to_bitfields?(t) ∧
no_pointer_to_member_to_reference?(t) ∧
no_pointer_to_member_to_cv_void?(t) ∧
no_cv_void_parameter?(t) ∧
no_array_of_references?(t) ∧
no_array_of_cv_void?(t) ∧
no_array_of_function?(t) ∧
no_array_of_abstract_class?(t) ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧
no_array_return_type?(t) ∧
no_function_return_type?(t) ∧
bitfield_underlying_integral_or_enum_type?(t) ∧
cv_array?(t) ∧
enum_underlying_integral?(t) ∧
enum_constants?(t) ∧
const_volatile?(t) ∧
const_stutter?(t) ∧
volatile_stutter?(t) ∧
no_cv_class?(t) ∧
no_cv_union?(t) ∧
no_cv_function?(t) ∧
no_reference_to_void?(t) ∧
no_cv_void?(t) ∧ no_array_of_bitfields?(t)
{-2} const?(cv_base(typ(typ(typ′))))
{-3} volatile?(typ(typ′))
{-4} volatile?(typ′)

Instantiating quantified variables,
Expanding the definition of volatile_stutter?,
Expanding the definition of subterm,
which is trivially true.
This completes the proof of `cv_base_not_const.4`.
Q.E.D.

C.40.27 Cpp_Deep_Type_Wellformedness.cv_base_not_volatile

Terse proof for `cv_base_not_volatile`.

`cv_base_not_volatile:`

{1} ∀ (typ: Cpp_Type): ¬ volatile?(cv_base(typ))

Repeatedly Skolemizing and flattening,
Expanding the definition of cv_base,
Expanding the definition of cv_base,

Expanding the definition of Cpp_Type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

`cv_base_not_volatile.1:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ$'$)

{-3} const?(typ(typ$'$))

{-4} volatile?(cv_base(typ(typ(typ$'$))))

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of const_stutter?,

which is trivially true.

This completes the proof of `cv_base_not_volatile.1`.

`cv_base_not_volatile.2:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ′) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ′)

{-3} volatile?(cv_base(typ(typ(typ′))))

{-4} volatile?(typ(typ′))

Expanding the definition of cv_base,

Instantiating quantified variables,

Expanding the definition of subterm,

Expanding the definition of subterm,

Expanding the definition of volatile_stutter?,

Expanding the definition of const_volatile?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_base_not_volatile.2`.

`cv_base_not_volatile.3:`

{-1} $\forall$ (t: Cpp_Type_):
subterm(t, typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)

{-2} const?(typ(typ$'$))

{-3} volatile?(cv_base(typ(typ(typ$'$))))

{-4} volatile?(typ$'$)

Instantiating quantified variables,

Expanding the definition of const_volatile?,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `cv_base_not_volatile.3`.

cv_base_not_volatile.4:

{-1} ∀ (t: Cpp_Type_):
subterm(t, typ′) ⊃
no_pointers_to_bitfield?(t) ∧
no_pointers_to_references?(t) ∧
no_cv_references?(t) ∧
no_reference_to_reference?(t) ∧
no_reference_to_bitfields?(t) ∧
no_pointer_to_member_to_reference?(t) ∧
no_pointer_to_member_to_cv_void?(t) ∧
no_cv_void_parameter?(t) ∧
no_array_of_references?(t) ∧
no_array_of_cv_void?(t) ∧
no_array_of_function?(t) ∧
no_array_of_abstract_class?(t) ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t) ∧
no_array_return_type?(t) ∧
no_function_return_type?(t) ∧
bitfield_underlying_integral_or_enum_type?(t) ∧
cv_array?(t) ∧
enum_underlying_integral?(t) ∧
enum_constants?(t) ∧
const_volatile?(t) ∧
const_stutter?(t) ∧
volatile_stutter?(t) ∧
no_cv_class?(t) ∧
no_cv_union?(t) ∧
no_cv_function?(t) ∧
no_reference_to_void?(t) ∧
no_cv_void?(t) ∧ no_array_of_bitfields?(t)
{-2} volatile?(cv_base(typ(typ(typ′))))
{-3} volatile?(typ(typ′))
{-4} volatile?(typ′)

Instantiating quantified variables,
Expanding the definition of volatile_stutter?,
Expanding the definition of subterm,
which is trivially true.
This completes the proof of cv_base_not_volatile.4.
Q.E.D.

C.40.28 Cpp_Deep_Type_Wellformedness.cv_base_cv

Terse proof for cv_base_cv.

cv_base_cv:

{1} ∀ (P: PRED[Cpp_Type_], typ: Cpp_Type):
(P ⊆ interpreted?) ⊃ (P(cv_base(typ)) ≡ cv(P)(typ))

Repeatedly Skolemizing and flattening,
Applying propositional simplification,

we get 2 subgoals:
`cv_base_cv.1:`

{-1}	P'(cv_base(typ$'$))
{-2}	Cpp_Type?(typ$'$)
{-3}	($P' \subseteq$ interpreted?)
{1}	cv(P')(typ$'$)

Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of cv_base,
Installing automatic rewrites from: subterm const_stutter? volatile_stutter? const_volatile?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 5 subgoals:
`cv_base_cv.1.1:`

{-1}	const?(typ$'$)
{-2}	P'(cv_base(typ(typ$'$)))
{-3}	Cpp_Type?(typ$'$)
{-4}	($P' \subseteq$ interpreted?)
{1}	P'(typ$'$)
{2}	P'(typ(typ$'$))
{3}	P'(typ(typ(typ$'$)))

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`cv_base_cv.1.1.1:`

{-1}	const?(typ$'$)
{-2}	const?(typ(typ$'$))
{-3}	P'(cv_base(typ(typ(typ$'$))))
{-4}	Cpp_Type?(typ$'$)
{-5}	($P' \subseteq$ interpreted?)
{1}	P'(typ(typ$'$))
{2}	P'(typ$'$)
{3}	P'(typ(typ(typ$'$)))

Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.1.1`.
`cv_base_cv.1.1.2:`

{-1}	const?(typ$'$)
{-2}	P'(cv_base(typ(typ(typ$'$))))
{-3}	volatile?(typ(typ$'$))
{-4}	Cpp_Type?(typ$'$)
{-5}	($P' \subseteq$ interpreted?)
{1}	P'(typ(typ$'$))
{2}	P'(typ$'$)
{3}	P'(typ(typ(typ$'$)))

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`cv_base_cv.1.1.2.1:`

{-1}	const?(typ′)
{-2}	const?(typ(typ(typ′)))
{-3}	P'(cv_base(typ(typ(typ(typ′)))))
{-4}	volatile?(typ(typ′))
{-5}	Cpp_Type?(typ′)
{-6}	($P' \subseteq$ interpreted?)
{1}	P'(typ(typ(typ′)))
{2}	P'(typ(typ′))
{3}	P'(typ′)

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_base_cv.1.1.2.1`.

`cv_base_cv.1.1.2.2:`

{-1}	const?(typ′)
{-2}	P'(cv_base(typ(typ(typ(typ′)))))
{-3}	volatile?(typ(typ(typ′)))
{-4}	volatile?(typ(typ′))
{-5}	Cpp_Type?(typ′)
{-6}	($P' \subseteq$ interpreted?)
{1}	P'(typ(typ(typ′)))
{2}	P'(typ(typ′))
{3}	P'(typ′)

Expanding the definition of Cpp_Type?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_base_cv.1.1.2.2`.

`cv_base_cv.1.2:`

{-1}	P'(cv_base(typ(typ′)))
{-2}	volatile?(typ′)
{-3}	Cpp_Type?(typ′)
{-4}	($P' \subseteq$ interpreted?)
{1}	P'(typ′)
{2}	P'(typ(typ′))
{3}	P'(typ(typ(typ′)))

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`cv_base_cv.1.2.1:`

{-1}	const?(typ(typ′))
{-2}	P'(cv_base(typ(typ(typ′))))
{-3}	volatile?(typ′)
{-4}	Cpp_Type?(typ′)
{-5}	($P' \subseteq$ interpreted?)
{1}	P'(typ(typ′))
{2}	P'(typ′)
{3}	P'(typ(typ(typ′)))

Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.2.1`.

`cv_base_cv.1.2.2:`

{-1}	P'(cv_base(typ(typ(typ′))))
{-2}	volatile?(typ(typ′))
{-3}	volatile?(typ′)
{-4}	Cpp_Type?(typ′)
{-5}	($P' \subseteq$ interpreted?)
{1}	P'(typ(typ′))
{2}	P'(typ′)
{3}	P'(typ(typ(typ′)))

Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.2.2`.

`cv_base_cv.1.3:`

{-1}	const?(typ′)
{-2}	P'(cv_base(typ(typ′)))
{-3}	Cpp_Type?(typ′)
{-4}	($P' \subseteq$ interpreted?)
{1}	P'(typ′)
{2}	P'(typ(typ′))
{3}	volatile?(typ(typ′))

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.3`.

`cv_base_cv.1.4:`

{-1}	P'(cv_base(typ(typ′)))
{-2}	volatile?(typ′)
{-3}	Cpp_Type?(typ′)
{-4}	($P' \subseteq$ interpreted?)
{1}	P'(typ′)
{2}	P'(typ(typ′))
{3}	volatile?(typ(typ′))

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.4`.
`cv_base_cv.1.5:`

{-1}	$P'(\text{cv\_base}(\text{typ}(\text{typ}')))$
{-2}	volatile?(typ′)
{-3}	Cpp_Type?(typ′)
{-4}	($P' \subseteq$ interpreted?)
{1}	$P'(\text{typ}')$
{2}	$P'(\text{typ}(\text{typ}'))$

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`cv_base_cv.1.5.1:`

{-1}	const?(typ(typ′))
{-2}	$P'(\text{cv\_base}(\text{typ}(\text{typ}(\text{typ}'))))$
{-3}	volatile?(typ′)
{-4}	Cpp_Type?(typ′)
{-5}	($P' \subseteq$ interpreted?)
{1}	$P'(\text{typ}(\text{typ}'))$
{2}	$P'(\text{typ}')$

Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.5.1`.
`cv_base_cv.1.5.2:`

{-1}	$P'(\text{cv\_base}(\text{typ}(\text{typ}(\text{typ}'))))$
{-2}	volatile?(typ(typ′))
{-3}	volatile?(typ′)
{-4}	Cpp_Type?(typ′)
{-5}	($P' \subseteq$ interpreted?)
{1}	$P'(\text{typ}(\text{typ}'))$
{2}	$P'(\text{typ}')$

Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_base_cv.1.5.2`.
`cv_base_cv.2:`

{-1}	$\text{cv}(P')(\text{typ}')$
{-2}	Cpp_Type?(typ′)
{-3}	($P' \subseteq$ interpreted?)
{1}	$P'(\text{cv\_base}(\text{typ}'))$

Rewriting using cv_base_result, matching in * where P gets P!1,
This completes the proof of `cv_base_cv.2`.
Q.E.D.

C.40.29 Cpp_Deep_Type_Wellformedness.layout_compatible?_TCC1

Terse proof for `layout_compatible?_TCC1`.

`layout_compatible?_TCC1:`

{1} ∃ (x: equivalence[Cpp_Type]): TRUE

Instantiating the top quantifier in 1 with the terms: (=),
we get 2 subgoals:

`layout_compatible?_TCC1.1:`

{1} TRUE

which is trivially true.
This completes the proof of `layout_compatible?_TCC1.1`.

`layout_compatible?_TCC1.2:`

{1} equivalence?[Cpp_Type]
(restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean](=))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `layout_compatible?_TCC1.2`.
Q.E.D.

C.40.30 Cpp_Deep_Type_Wellformedness.layout_compatible_enums_TCC1

Terse proof for `layout_compatible_enums_TCC1`.

`layout_compatible_enums_TCC1:`

{1} ∀ (e_1, e_2: Cpp_Subtype(enum?)):
array?(e_1) ∨
pointer?(e_1) ∨
reference?(e_1) ∨
bitfield?(e_1) ∨ enum?(e_1) ∨ pointer_to_member?(e_1) ∨ const?(e_1) ∨ volatile?(e_1)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `layout_compatible_enums_TCC1`.
Q.E.D.

C.40.31 Cpp_Deep_Type_Wellformedness.layout_compatible_enums_TCC2

Terse proof for `layout_compatible_enums_TCC2`.

`layout_compatible_enums_TCC2:`

{1} ∀ (e_1, e_2: Cpp_Subtype(enum?)):
array?(e_2) ∨
pointer?(e_2) ∨
reference?(e_2) ∨
bitfield?(e_2) ∨ enum?(e_2) ∨ pointer_to_member?(e_2) ∨ const?(e_2) ∨ volatile?(e_2)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `layout_compatible_enums_TCC2`.
Q.E.D.

C.41 Proofs for Cpp_Deep_Types (types.pvs)

C.41.1 Cpp_Deep_Types.Cpp_Type__pointer_to_member_eta_TCC1

Terse proof for `Cpp_Type__pointer_to_member_eta_TCC1`.
`Cpp_Type__pointer_to_member_eta_TCC1:`

```
{1}  ∀ (pointer_to_member?_var: (pointer_to_member?)):
       array?(pointer_to_member?_var) ∨
         pointer?(pointer_to_member?_var) ∨
           reference?(pointer_to_member?_var) ∨
             bitfield?(pointer_to_member?_var) ∨
               enum?(pointer_to_member?_var) ∨
                 pointer_to_member?(pointer_to_member?_var) ∨
                   const?(pointer_to_member?_var) ∨ volatile?(pointer_to_member?_var)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Cpp_Type__pointer_to_member_eta_TCC1`.
Q.E.D.

C.41.2 Cpp_Deep_Types.Cpp_Type__pointer_to_member_eta_TCC2

Terse proof for `Cpp_Type__pointer_to_member_eta_TCC2`.
`Cpp_Type__pointer_to_member_eta_TCC2:`

```
{1}  ∀ (pointer_to_member?_var: (pointer_to_member?)): class?(typ(pointer_to_member?_var))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for typ(pointer_to_member?_var!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `Cpp_Type__pointer_to_member_eta_TCC2`.
Q.E.D.

C.41.3 Cpp_Deep_Types.Cpp_Type__typ_pointer_to_member_TCC1

Terse proof for `Cpp_Type__typ_pointer_to_member_TCC1`.
`Cpp_Type__typ_pointer_to_member_TCC1:`

```
{1}  ∀ (pointer_to_member1_var: (class?), pointer_to_member2_var: Cpp_Type_):
       array?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨
         pointer?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨
           reference?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨
             bitfield?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨
               enum?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨
                 pointer_to_member?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var
                   const?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∨
                     volatile?(pointer_to_member(pointer_to_member1_var, pointer_to_member2_var))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Cpp_Type__typ_pointer_to_member_TCC1`.
Q.E.D.

C.41.4 Cpp_Deep_Types.subterm_less

Terse proof for `subterm_less`.
`subterm_less:`

{1} $\forall$ (t_1, t_2: Cpp_Type_): subterm(t_1, t_2) $\wedge$ $t_1 \neq t_2 \supset t_1 \ll t_2$

Inducting on t_2 on formula 1,
we get 28 subgoals:
`subterm_less.1:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, uchar) $\wedge$ $t_1 \neq$ uchar $\supset t_1 \ll$ uchar

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.1`.
`subterm_less.2:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, schar) $\wedge$ $t_1 \neq$ schar $\supset t_1 \ll$ schar

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.2`.
`subterm_less.3:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, char) $\wedge$ $t_1 \neq$ char $\supset t_1 \ll$ char

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.3`.
`subterm_less.4:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, short) $\wedge$ $t_1 \neq$ short $\supset t_1 \ll$ short

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.4`.
`subterm_less.5:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, int) $\wedge$ $t_1 \neq$ int $\supset t_1 \ll$ int

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.5`.
`subterm_less.6:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, long) $\wedge$ $t_1 \neq$ long $\supset t_1 \ll$ long

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.6`.
`subterm_less.7:`

{1} $\forall$ (t_1: Cpp_Type_): subterm(t_1, longlong) $\wedge$ $t_1 \neq$ longlong $\supset t_1 \ll$ longlong

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.7`.
`subterm_less.8:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{ushort}) \wedge t_1 \neq \text{ushort} \supset t_1 \ll \text{ushort}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.8`.
`subterm_less.9:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{uint}) \wedge t_1 \neq \text{uint} \supset t_1 \ll \text{uint}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.9`.
`subterm_less.10:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{ulong}) \wedge t_1 \neq \text{ulong} \supset t_1 \ll \text{ulong}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.10`.
`subterm_less.11:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{ulonglong}) \wedge t_1 \neq \text{ulonglong} \supset t_1 \ll \text{ulonglong}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.11`.
`subterm_less.12:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{wchar\_t}) \wedge t_1 \neq \text{wchar\_t} \supset t_1 \ll \text{wchar\_t}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.12`.
`subterm_less.13:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{bool}) \wedge t_1 \neq \text{bool} \supset t_1 \ll \text{bool}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.13`.
`subterm_less.14:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{float}) \wedge t_1 \neq \text{float} \supset t_1 \ll \text{float}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.14`.
`subterm_less.15:`

$$\{1\} \quad \forall\ (t_1 \texttt{:}\ \text{Cpp\_Type\_})\texttt{:}\ \text{subterm}(t_1\texttt{,}\ \text{double}) \wedge t_1 \neq \text{double} \supset t_1 \ll \text{double}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.15`.

`subterm_less.16:`

{1} ∀ (t_1: Cpp_Type_): subterm(t_1, longdouble) ∧ t_1 ≠ longdouble ⊃ t_1 ≪ longdouble

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.16`.
`subterm_less.17:`

{1} ∀ (t_1: Cpp_Type_): subterm(t_1, void) ∧ t_1 ≠ void ⊃ t_1 ≪ void

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.17`.
`subterm_less.18:`

{1} ∀ (array_type1_var: Cpp_Type_, array_type2_var: nat):
(∀ (t_1: Cpp_Type_):
subterm(t_1, array_type1_var) ∧ t_1 ≠ array_type1_var ⊃ t_1 ≪ array_type1_var)
⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, array_type(array_type1_var, array_type2_var)) ∧
t_1 ≠ array_type(array_type1_var, array_type2_var)
⊃ t_1 ≪ array_type(array_type1_var, array_type2_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.18`.
`subterm_less.19:`

{1} ∀ (function_type1_var: Cpp_Type_, function_type2_var: list⌊Cpp_Type_⌋,
function_type3_var: bool):
(∀ (t_1: Cpp_Type_):
subterm(t_1, function_type1_var) ∧ t_1 ≠ function_type1_var ⊃ t_1 ≪ function_type1_var)
∧
every(λ (t_2: Cpp_Type_): ∀ (t_1: Cpp_Type_): subterm(t_1, t_2) ∧ t_1 ≠ t_2 ⊃ t_1 ≪ t_2)
(function_type2_var)
⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, function_type(function_type1_var, function_type2_var, function_type3_var))
∧ t_1 ≠ function_type(function_type1_var, function_type2_var, function_type3_var)
⊃ t_1 ≪ function_type(function_type1_var, function_type2_var, function_type3_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`subterm_less.19.1:`

{-1} $\forall\ (t_1\colon \text{Cpp\_Type\_})\colon$
$\quad \text{subterm}(t_1,\ \text{function\_type1\_var}') \wedge t_1 \neq \text{function\_type1\_var}' \supset t_1 \ll \text{function\_type1\_var}'$
{-2} $\text{every}(\lambda\ (t_2\colon \text{Cpp\_Type\_})\colon \forall\ (t_1\colon \text{Cpp\_Type\_})\colon \text{subterm}(t_1,\ t_2) \wedge t_1 \neq t_2 \supset t_1 \ll t_2)$
$\quad (\text{function\_type2\_var}')$
{-3} $\text{subterm}(t_1',\ \text{function\_type1\_var}')$

{1} $t_1' = \text{function\_type}(\text{function\_type1\_var}',\ \text{function\_type2\_var}',\ \text{function\_type3\_var}')$
{2} $t_1' = \text{function\_type1\_var}'$
{3} $t_1' \ll \text{function\_type1\_var}'$
{4} $\text{some}[\text{Cpp\_Type\_}](\lambda\ (z\colon \text{Cpp\_Type\_})\colon t_1' = z \vee t_1' \ll z)(\text{function\_type2\_var}')$

Instantiating the top quantifier in -1 with the terms: (t1!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.19.1`.
`subterm_less.19.2:`

{-1} $\forall\ (t_1\colon \text{Cpp\_Type\_})\colon$
$\quad \text{subterm}(t_1,\ \text{function\_type1\_var}') \wedge t_1 \neq \text{function\_type1\_var}' \supset t_1 \ll \text{function\_type1\_var}'$
{-2} $\text{every}(\lambda\ (t_2\colon \text{Cpp\_Type\_})\colon \forall\ (t_1\colon \text{Cpp\_Type\_})\colon \text{subterm}(t_1,\ t_2) \wedge t_1 \neq t_2 \supset t_1 \ll t_2)$
$\quad (\text{function\_type2\_var}')$
{-3} $\text{some}[\text{Cpp\_Type\_}](\lambda\ (z\colon \text{Cpp\_Type\_})\colon \text{subterm}(t_1',\ z))(\text{function\_type2\_var}')$

{1} $t_1' = \text{function\_type}(\text{function\_type1\_var}',\ \text{function\_type2\_var}',\ \text{function\_type3\_var}')$
{2} $t_1' = \text{function\_type1\_var}'$
{3} $t_1' \ll \text{function\_type1\_var}'$
{4} $\text{some}[\text{Cpp\_Type\_}](\lambda\ (z\colon \text{Cpp\_Type\_})\colon t_1' = z \vee t_1' \ll z)(\text{function\_type2\_var}')$

Rewriting using some_iff_exists, matching in *,
Rewriting using some_iff_exists, matching in *,
Rewriting using every_iff_forall, matching in *,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 4 with the terms: (t!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -2 with the terms: (t!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.19.2`.
`subterm_less.20:`

{1} $\forall\ (\text{pointer1\_var}\colon \text{Cpp\_Type\_})\colon$
$\quad (\forall\ (t_1\colon \text{Cpp\_Type\_})\colon \text{subterm}(t_1,\ \text{pointer1\_var}) \wedge t_1 \neq \text{pointer1\_var} \supset t_1 \ll \text{pointer1\_var})$
$\quad \supset$
$\quad (\forall\ (t_1\colon \text{Cpp\_Type\_})\colon$
$\qquad \text{subterm}(t_1,\ \text{pointer}(\text{pointer1\_var})) \wedge t_1 \neq \text{pointer}(\text{pointer1\_var}) \supset$
$\qquad t_1 \ll \text{pointer}(\text{pointer1\_var}))$

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Expanding the definition of subterm,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.20`.

`subterm_less.21:`

{1} ∀ (reference1_var: Cpp_Type_):
(∀ (t_1: Cpp_Type_):
subterm(t_1, reference1_var) ∧ t_1 ≠ reference1_var ⊃ t_1 ≪ reference1_var)
⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, reference(reference1_var)) ∧ t_1 ≠ reference(reference1_var) ⊃
t_1 ≪ reference(reference1_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Expanding the definition of subterm,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.21`.
`subterm_less.22:`

{1} ∀ (class1_var: string, class2_var: bool):
∀ (t_1: Cpp_Type_):
subterm(t_1, class(class1_var, class2_var)) ∧ t_1 ≠ class(class1_var, class2_var) ⊃
t_1 ≪ class(class1_var, class2_var)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.22`.
`subterm_less.23:`

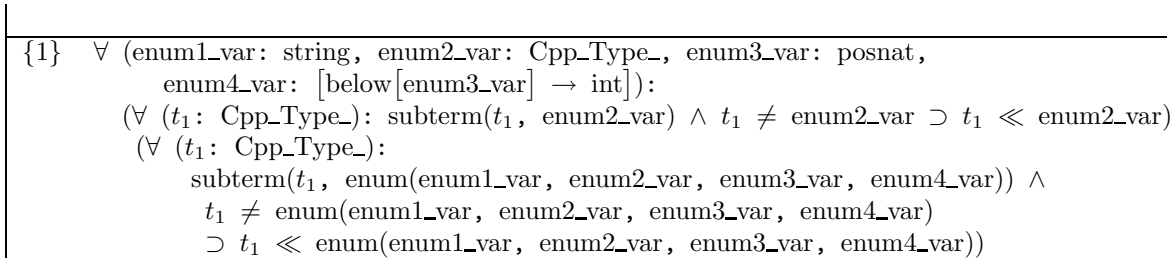
{1} ∀ (union1_var: string):
∀ (t_1: Cpp_Type_):
subterm(t_1, union(union1_var)) ∧ t_1 ≠ union(union1_var) ⊃ t_1 ≪ union(union1_var)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_less.23`.
`subterm_less.24:`

{1} ∀ (bitfield1_var: Cpp_Type_, bitfield2_var: nat, bitfield3_var: posnat):
(∀ (t_1: Cpp_Type_):
subterm(t_1, bitfield1_var) ∧ t_1 ≠ bitfield1_var ⊃ t_1 ≪ bitfield1_var)
⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, bitfield(bitfield1_var, bitfield2_var, bitfield3_var)) ∧
t_1 ≠ bitfield(bitfield1_var, bitfield2_var, bitfield3_var)
⊃ t_1 ≪ bitfield(bitfield1_var, bitfield2_var, bitfield3_var))

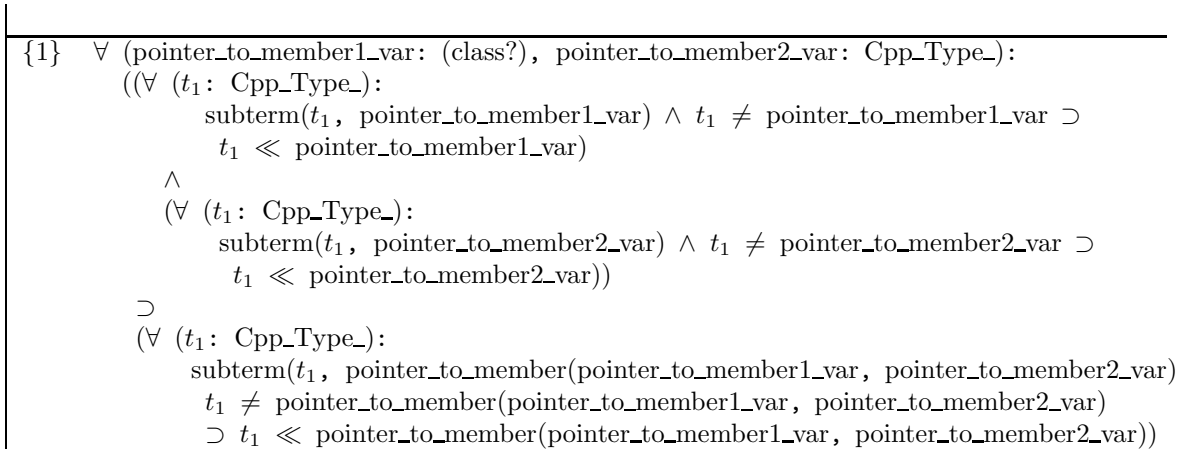
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Expanding the definition of ■,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.24`.

`subterm_less.25:`



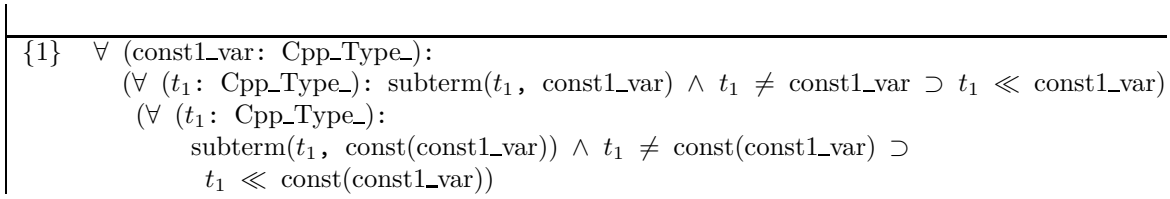
{1} ∀ (enum1_var: string, enum2_var: Cpp_Type_, enum3_var: posnat,
enum4_var: [below[enum3_var] → int]):
(∀ (t_1: Cpp_Type_): subterm(t_1, enum2_var) ∧ t_1 ≠ enum2_var ⊃ t_1 ≪ enum2_var) ⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, enum(enum1_var, enum2_var, enum3_var, enum4_var)) ∧
t_1 ≠ enum(enum1_var, enum2_var, enum3_var, enum4_var)
⊃ t_1 ≪ enum(enum1_var, enum2_var, enum3_var, enum4_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Expanding the definition of ■,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.25`.
`subterm_less.26:`



{1} ∀ (pointer_to_member1_var: (class?), pointer_to_member2_var: Cpp_Type_):
((∀ (t_1: Cpp_Type_):
subterm(t_1, pointer_to_member1_var) ∧ t_1 ≠ pointer_to_member1_var ⊃
t_1 ≪ pointer_to_member1_var)
∧
(∀ (t_1: Cpp_Type_):
subterm(t_1, pointer_to_member2_var) ∧ t_1 ≠ pointer_to_member2_var ⊃
t_1 ≪ pointer_to_member2_var))
⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)) ∧
t_1 ≠ pointer_to_member(pointer_to_member1_var, pointer_to_member2_var)
⊃ t_1 ≪ pointer_to_member(pointer_to_member1_var, pointer_to_member2_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Expanding the definition of subterm,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -2 with the terms: (t1!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.26`.
`subterm_less.27:`



{1} ∀ (const1_var: Cpp_Type_):
(∀ (t_1: Cpp_Type_): subterm(t_1, const1_var) ∧ t_1 ≠ const1_var ⊃ t_1 ≪ const1_var) ⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, const(const1_var)) ∧ t_1 ≠ const(const1_var) ⊃
t_1 ≪ const(const1_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Expanding the definition of subterm,
Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.27`.
`subterm_less.28`:

{1} $\forall$ (volatile1_var: Cpp_Type_):
(∀ (t_1: Cpp_Type_):
subterm(t_1, volatile1_var) ∧ $t_1 \neq$ volatile1_var ⊃ $t_1 \ll$ volatile1_var)
⊃
(∀ (t_1: Cpp_Type_):
subterm(t_1, volatile(volatile1_var)) ∧ $t_1 \neq$ volatile(volatile1_var) ⊃
$t_1 \ll$ volatile(volatile1_var))

Repeatedly Skolemizing and flattening,
Expanding the definition of ■,
Expanding the definition of subterm,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_less.28`.
Q.E.D.

C.41.5 Cpp_Deep_Types.subterm_transitive

Terse proof for `subterm_transitive`.
`subterm_transitive`:

{1} ∀ (t_1, t_2, t_3: Cpp_Type_): subterm(t_1, t_2) ∧ subterm(t_2, t_3) ⊃ subterm(t_1, t_3)

Using lemma wf_induction[Cpp_Type_, ≪],
For the top quantifier in 1, we introduce Skolem constants: (t1!1 t2!1 _),
Instantiating the top quantifier in -1 with the terms: (LAMBDA (t3: Cpp_Type_): subterm(t1!1, t2!1) AND subterm(t2!1, t3) IMPLIES subterm(t1!1, t3)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Splitting conjunctions,
we get 3 subgoals:
`subterm_transitive.1`:

{-1} $t'_2 = x'$
{-2} ∀ (y: Cpp_Type_):
$y \ll x' \supset$ subterm(t'_1, t'_2) ∧ subterm(t'_2, y) ⊃ subterm(t'_1, y)
{-3} subterm(t'_1, t'_2)

{1} subterm(t'_1, x')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `subterm_transitive.1`.
`subterm_transitive.2`:

{-1} ∀ (y: Cpp_Type_):
$y \ll x' \supset$ subterm(t'_1, t'_2) ∧ subterm(t'_2, y) ⊃ subterm(t'_1, y)
{-2} subterm(t'_1, t'_2)

{1} TRUE
{2} subterm(t'_1, x')

which is trivially true.

This completes the proof of `subterm_transitive.2`.

`subterm_transitive.3:`

{-1} ¬ uchar?(x') ∧
IF schar?(x') THEN FALSE
ELSIF char?(x') THEN FALSE
ELSIF short?(x') THEN FALSE
ELSIF int?(x') THEN FALSE
ELSIF long?(x') THEN FALSE
ELSIF longlong?(x') THEN FALSE
ELSIF ushort?(x') THEN FALSE
ELSIF uint?(x') THEN FALSE
ELSIF ulong?(x') THEN FALSE
ELSIF ulonglong?(x') THEN FALSE
ELSIF wchar_t?(x') THEN FALSE
ELSIF bool?(x') THEN FALSE
ELSIF float?(x') THEN FALSE
ELSIF double?(x') THEN FALSE
ELSIF longdouble?(x') THEN FALSE
ELSIF void?(x') THEN FALSE
ELSIF array?(x') THEN subterm(t'_2, typ(x'))
ELSIF function?(x')
THEN subterm(t'_2, ret_typ(x')) ∨
some[Cpp_Type_](λ (z: Cpp_Type_): subterm(t'_2, z))(args_typ(x'))
ELSIF pointer?(x') THEN subterm(t'_2, typ(x'))
ELSIF reference?(x') THEN subterm(t'_2, typ(x'))
ELSIF class?(x') THEN FALSE
ELSIF union?(x') THEN FALSE
ELSIF bitfield?(x') THEN subterm(t'_2, typ(x'))
ELSIF enum?(x') THEN subterm(t'_2, typ(x'))
ELSIF pointer_to_member?(x')
THEN subterm(t'_2, typ(x')) ∨ subterm(t'_2, member_typ(x'))
ELSIF const?(x') THEN subterm(t'_2, typ(x'))
ELSE subterm(t'_2, typ(x'))

{-2} ∀ (y: Cpp_Type_):
$y \ll x' \supset$ subterm(t'_1, t'_2) ∧ subterm(t'_2, y) ⊃ subterm(t'_1, y)

{-3} subterm(t'_1, t'_2)

{1} subterm(t'_1, x')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 7 subgoals:

`subterm_transitive.3.1:`

{-1} array?(x')
{-2} subterm(t'_2, typ(x'))
{-3} ∀ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t'_2, y) ⊃ subterm(t'_1, y)
{-4} subterm(t'_1, t'_2)

{1} subterm(t'_1, x')

Instantiating the top quantifier in -3 with the terms: (typ(x!1)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`subterm_transitive.3.1.1:`

{-1}	array?(x')
{-2}	subterm(t_2', typ(x'))
{-3}	subterm(t_1', typ(x'))
{-4}	subterm(t_1', t_2')
{1}	subterm(t_1', x')

Expanding the definition of subterm,
which is trivially true.
This completes the proof of `subterm_transitive.3.1.1`.
`subterm_transitive.3.1.2:`

{-1}	array?(x')
{-2}	subterm(t_2', typ(x'))
{-3}	subterm(t_1', t_2')
{1}	typ(x') $\ll$ x'
{2}	subterm(t_1', x')

Expanding the definition of ■,
which is trivially true.
This completes the proof of `subterm_transitive.3.1.2`.
`subterm_transitive.3.2:`

{-1}	function?(x')
{-2}	subterm(t_2', ret_typ(x'))
{-3}	$\forall$ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t_2', y) $\supset$ subterm(t_1', y)
{-4}	subterm(t_1', t_2')
{1}	subterm(t_1', x')

Instantiating the top quantifier in -3 with the terms: (ret_typ(x!1)),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_transitive.3.2`.
`subterm_transitive.3.3:`

{-1}	function?(x')
{-2}	some[Cpp_Type_](λ (z: Cpp_Type_): subterm(t_2', z))(args_typ(x'))
{-3}	$\forall$ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t_2', y) $\supset$ subterm(t_1', y)
{-4}	subterm(t_1', t_2')
{1}	subterm(t_1', x')

Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Rewriting using some_iff_exists, matching in *,
Rewriting using some_iff_exists, matching in *,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ■,
Applying disjunctive simplification to flatten sequent,
Rewriting using some_iff_exists, matching in *,
Using lemma forall_not[Cpp_Type_],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -1 with the terms: (x!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:
`subterm_transitive.3.3.1:`

{-1}	function?(x')
{-2}	member(t', args_typ(x'))
{-3}	subterm(t'_2, t')
{-4}	subterm(t'_1, t'_2)
{1}	$t' = x'$
{2}	$t' \ll x'$
{3}	$\exists$ (t: Cpp_Type_): member(t, args_typ(x')) $\wedge$ ($t' = t \vee t' \ll t$)
{4}	t' = ret_typ(x')
{5}	$t' \ll$ ret_typ(x')
{6}	subterm(t'_1, t')
{7}	$t'_1 = x'$
{8}	subterm(t'_1, ret_typ(x'))

Using lemma subterm_less,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
This completes the proof of `subterm_transitive.3.3.1`.
`subterm_transitive.3.3.2:`

{-1}	function?(x')
{-2}	member(t', args_typ(x'))
{-3}	subterm(t'_2, t')
{-4}	subterm(t'_1, t'_2)
{1}	member(x', args_typ(x'))
{2}	$\exists$ (t: Cpp_Type_): member(t, args_typ(x')) $\wedge$ ($t' = t \vee t' \ll t$)
{3}	t' = ret_typ(x')
{4}	$t' \ll$ ret_typ(x')
{5}	subterm(t'_1, t')
{6}	$t'_1 = x'$
{7}	subterm(t'_1, ret_typ(x'))

Instantiating the top quantifier in 2 with the terms: (t!1),
which is trivially true.
This completes the proof of `subterm_transitive.3.3.2`.
`subterm_transitive.3.4:`

{-1}	subterm(t'_2, typ(x'))
{-2}	pointer?(x')
{-3}	$\forall$ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t'_2, y) $\supset$ subterm(t'_1, y)
{-4}	subterm(t'_1, t'_2)
{1}	subterm(t'_1, x')

Instantiating the top quantifier in -3 with the terms: (typ(x!1)),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_transitive.3.4`.
`subterm_transitive.3.5:`

{-1}	subterm(t'_2, typ(x'))
{-2}	reference?(x')
{-3}	$\forall$ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t'_2, y) $\supset$ subterm(t'_1, y)
{-4}	subterm(t'_1, t'_2)
{1}	subterm(t'_1, x')

Instantiating the top quantifier in -3 with the terms: (typ(x!1)),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_transitive.3.5`.
`subterm_transitive.3.6`:

{-1}	subterm(t'_2, typ(x'))
{-2}	$\forall$ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t'_2, y) $\supset$ subterm(t'_1, y)
{-3}	subterm(t'_1, t'_2)
{1}	uchar?(x')
{2}	schar?(x')
{3}	char?(x')
{4}	short?(x')
{5}	int?(x')
{6}	long?(x')
{7}	longlong?(x')
{8}	ushort?(x')
{9}	uint?(x')
{10}	ulong?(x')
{11}	ulonglong?(x')
{12}	wchar_t?(x')
{13}	bool?(x')
{14}	float?(x')
{15}	double?(x')
{16}	longdouble?(x')
{17}	void?(x')
{18}	function?(x')
{19}	class?(x')
{20}	union?(x')
{21}	subterm(t'_1, x')

Instantiating the top quantifier in -2 with the terms: (typ(x!1)),
Expanding the definition of ■,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subterm_transitive.3.6`.
`subterm_transitive.3.7`:

{-1}	pointer_to_member?(x')
{-2}	subterm(t'_2, member_typ(x'))
{-3}	$\forall$ (y: Cpp_Type_): $y \ll x' \supset$ subterm(t'_2, y) $\supset$ subterm(t'_1, y)
{-4}	subterm(t'_1, t'_2)
{1}	subterm(t'_1, x')

Instantiating the top quantifier in -3 with the terms: (member_typ(x!1)),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subterm_transitive.3.7`.
Q.E.D.

C.41.6 Cpp_Deep_Types.c_TCC1

Terse proof for `c_TCC1`.

c_TCC1:

{1} ∀ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
const?(typ) ∧ ¬ p(typ) ⊃
array?(typ) ∨
pointer?(typ) ∨
reference?(typ) ∨
bitfield?(typ) ∨
enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of c_TCC1.
Q.E.D.

C.41.7 Cpp_Deep_Types.v_TCC1

Terse proof for v_TCC1.

v_TCC1:

{1} ∀ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
volatile?(typ) ∧ ¬ p(typ) ⊃
array?(typ) ∨
pointer?(typ) ∨
reference?(typ) ∨
bitfield?(typ) ∨
enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of v_TCC1.
Q.E.D.

C.41.8 Cpp_Deep_Types.cv_TCC1

Terse proof for cv_TCC1.

cv_TCC1:

{1} ∀ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
const?(typ) ∧ ¬ v(p)(typ) ∧ ¬ c(p)(typ) ⊃
array?(typ) ∨
pointer?(typ) ∨
reference?(typ) ∨
bitfield?(typ) ∨
enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of cv_TCC1.
Q.E.D.

C.41.9 Cpp_Deep_Types.cv_TCC2

Terse proof for cv_TCC2.

`cv_TCC2:`

{1} ∀ (p: PRED[Cpp_Type_], typ: Cpp_Type_):
volatile?(typ(typ)) ∧ const?(typ) ∧ ¬ $v(p)$(typ) ∧ ¬ $c(p)$(typ) ⊃
array?(typ(typ)) ∨
pointer?(typ(typ)) ∨
reference?(typ(typ)) ∨
bitfield?(typ(typ)) ∨
enum?(typ(typ)) ∨
pointer_to_member?(typ(typ)) ∨ const?(typ(typ)) ∨ volatile?(typ(typ))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cv_TCC2`.
Q.E.D.

C.41.10 Cpp_Deep_Types.cv_implied

Terse proof for `cv_implied`.

`cv_implied:`

{1} ∀ (p_1, p_2: PRED[Cpp_Type_], typ: Cpp_Type_):
$(p_1 \subseteq p_2)$ ∧ cv(p_1)(typ) ⊃ cv(p_2)(typ)

Repeatedly Skolemizing and flattening,
Expanding the definition of subset?,
Expanding the definition of member,
Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,
Applying disjunctive simplification to flatten sequent,
Splitting conjunctions,
we get 5 subgoals:

`cv_implied.1:`

{-1} p_1'(typ′)
{-2} ∀ (x: Cpp_Type_): $p_1'(x)$ ⇒ $p_2'(x)$

{1} p_2'(typ′)
{2} (const?(typ′) ∧ p_2'(typ(typ′)))
{3} p_2'(typ′)
{4} (volatile?(typ′) ∧ p_2'(typ(typ′)))
{5} (const?(typ′) ∧ volatile?(typ(typ′)) ∧ p_2'(typ(typ(typ′))))

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `cv_implied.1`.

`cv_implied.2:`

{-1} (const?(typ′) ∧ p_1'(typ(typ′)))
{-2} ∀ (x: Cpp_Type_): $p_1'(x)$ ⇒ $p_2'(x)$

{1} p_2'(typ′)
{2} (const?(typ′) ∧ p_2'(typ(typ′)))
{3} p_2'(typ′)
{4} (volatile?(typ′) ∧ p_2'(typ(typ′)))
{5} (const?(typ′) ∧ volatile?(typ(typ′)) ∧ p_2'(typ(typ(typ′))))

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.2`.

`cv_implied.3`:

{-1}	$p_1'(\text{typ}')$
{-2}	$\forall\ (x\colon\ \text{Cpp\_Type\_})\colon\ p_1'(x)\ \Rightarrow\ p_2'(x)$
{1}	$p_2'(\text{typ}')$
{2}	$(\text{const?}(\text{typ}')\ \wedge\ p_2'(\text{typ}(\text{typ}')))$
{3}	$p_2'(\text{typ}')$
{4}	$(\text{volatile?}(\text{typ}')\ \wedge\ p_2'(\text{typ}(\text{typ}')))$
{5}	$(\text{const?}(\text{typ}')\ \wedge\ \text{volatile?}(\text{typ}(\text{typ}'))\ \wedge\ p_2'(\text{typ}(\text{typ}(\text{typ}'))))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.3`.

`cv_implied.4`:

{-1}	$(\text{volatile?}(\text{typ}')\ \wedge\ p_1'(\text{typ}(\text{typ}')))$
{-2}	$\forall\ (x\colon\ \text{Cpp\_Type\_})\colon\ p_1'(x)\ \Rightarrow\ p_2'(x)$
{1}	$p_2'(\text{typ}')$
{2}	$(\text{const?}(\text{typ}')\ \wedge\ p_2'(\text{typ}(\text{typ}')))$
{3}	$p_2'(\text{typ}')$
{4}	$(\text{volatile?}(\text{typ}')\ \wedge\ p_2'(\text{typ}(\text{typ}')))$
{5}	$(\text{const?}(\text{typ}')\ \wedge\ \text{volatile?}(\text{typ}(\text{typ}'))\ \wedge\ p_2'(\text{typ}(\text{typ}(\text{typ}'))))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.4`.

`cv_implied.5`:

{-1}	$(\text{const?}(\text{typ}')\ \wedge\ \text{volatile?}(\text{typ}(\text{typ}'))\ \wedge\ p_1'(\text{typ}(\text{typ}(\text{typ}'))))$
{-2}	$\forall\ (x\colon\ \text{Cpp\_Type\_})\colon\ p_1'(x)\ \Rightarrow\ p_2'(x)$
{1}	$p_2'(\text{typ}')$
{2}	$(\text{const?}(\text{typ}')\ \wedge\ p_2'(\text{typ}(\text{typ}')))$
{3}	$p_2'(\text{typ}')$
{4}	$(\text{volatile?}(\text{typ}')\ \wedge\ p_2'(\text{typ}(\text{typ}')))$
{5}	$(\text{const?}(\text{typ}')\ \wedge\ \text{volatile?}(\text{typ}(\text{typ}'))\ \wedge\ p_2'(\text{typ}(\text{typ}(\text{typ}'))))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `cv_implied.5`.

Q.E.D.

C.41.11 Cpp_Deep_Types.pod?_TCC1

Terse proof for `pod?_TCC1`.

`pod?_TCC1:`

{1} ∀ (t: Cpp_Type_):
array?(t) ∧
¬ (class?(t) ∧ (pod_struct?(t) ∨ pod_union?(t))) ∧ ¬ scalar?(t)
⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Repeatedly Skolemizing and flattening,
This completes the proof of `pod?_TCC1`.
Q.E.D.

C.41.12 Cpp_Deep_Types.uidt_void_TCC1

Terse proof for `uidt_void_TCC1`.

`uidt_void_TCC1:`

{1} uninterpreted_data_type?((#size := 0, valid? := λ (l: list[Byte], a: Address): FALSE#))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_void_TCC1`.
Q.E.D.

C.42 Proofs for Cpp_Examples (cpp-examples.pvs)

C.42.1 Cpp_Examples.spec01

Terse proof for `spec01`.

`spec01:`

{1} Valid(STATES, skip[State], STATES)

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpressions.== not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.- Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference Aligned_Ptr plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read_single_data plain_memory_read_write_single_data pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr

plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s if_else switch member break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while do_while_unroll for_as_while for_unroll composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt allocate_stack deallocate_stack with_new_stackvar ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state l2r n2b a2b comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec01`.

Q.E.D.

C.42.2 Cpp_Examples.PRECONDITION_TCC1

Terse proof for `PRECONDITION_TCC1`.

`PRECONDITION_TCC1:`

{1} ∀ (*s*: State): pm‘states(*s*) ⊃ Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
PRECONDITION_TCC1.1:

{-1}	pm‘states(s')
{1}	Cpp_Type?(int)

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of PRECONDITION_TCC1.1.
PRECONDITION_TCC1.2:

{-1}	pm‘states(s')
{1}	cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of PRECONDITION_TCC1.2.
Q.E.D.

C.42.3 Cpp_Examples.PRECONDITION_TCC2

Terse proof for PRECONDITION_TCC2.
PRECONDITION_TCC2:

{1}	$\forall$ (s: State): in_blessed_memory?(dt_bool, c, pm‘rw_addr) $\wedge$ in_blessed_memory?(dt(int), m, pm‘rw_addr) $\wedge$ in_blessed_memory?(dt(int), n, pm‘rw_addr) $\wedge$ pm‘states(s) $\supset$ Cpp_Type?(char) $\wedge$ cv(non_bool_integral_enum?)(char)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
PRECONDITION_TCC2.1:

{-1}	in_blessed_memory?(dt_bool, c, pm‘rw_addr)
{-2}	in_blessed_memory?(dt(int), m, pm‘rw_addr)
{-3}	in_blessed_memory?(dt(int), n, pm‘rw_addr)
{-4}	pm‘states(s')
{1}	Cpp_Type?(char)

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields?

no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par
no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?
enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter?
no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Expanding the definition of Cpp_Type?,

Expanding the definition of subterm,

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `PRECONDITION_TCC2.1`.

`PRECONDITION_TCC2.2:`

{-1}	in_blessed_memory?(dt_bool, c, pm‘rw_addr)
{-2}	in_blessed_memory?(dt(int), m, pm‘rw_addr)
{-3}	in_blessed_memory?(dt(int), n, pm‘rw_addr)
{-4}	pm‘states(s')
{1}	cv(non_bool_integral_enum?)(char)

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `PRECONDITION_TCC2.2`.

Q.E.D.

C.42.4 Cpp_Examples.PRECONDITION_TCC3

Terse proof for `PRECONDITION_TCC3`.

`PRECONDITION_TCC3:`

{1} $\forall$ (s: State):
in_blessed_memory?(dt(char), b, pm‘rw_addr) $\wedge$
in_blessed_memory?(dt(char), a, pm‘rw_addr) $\wedge$
in_blessed_memory?(dt_bool, c, pm‘rw_addr) $\wedge$
in_blessed_memory?(dt(int), m, pm‘rw_addr) $\wedge$
in_blessed_memory?(dt(int), n, pm‘rw_addr) $\wedge$ pm‘states(s)
$\supset$ Cpp_Type?(char)

Repeatedly Skolemizing and flattening,

Using lemma PRECONDITION_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `PRECONDITION_TCC3`.

Q.E.D.

C.42.5 Cpp_Examples.PRECONDITION_TCC4

Terse proof for `PRECONDITION_TCC4`.

PRECONDITION_TCC4:

{1} ∀ (s: State):
blocks_disjoint?(b, size_of(char), a, size_of(char)) ∧
blocks_disjoint?(a, size_of(char), b, size_of(char)) ∧
in_blessed_memory?(dt(char), b, pm‘rw_addr) ∧
in_blessed_memory?(dt(char), a, pm‘rw_addr) ∧
in_blessed_memory?(dt_bool, c, pm‘rw_addr) ∧
in_blessed_memory?(dt(int), m, pm‘rw_addr) ∧
in_blessed_memory?(dt(int), n, pm‘rw_addr) ∧ pm‘states(s)
⊃ Cpp_Type?(int)

Repeatedly Skolemizing and flattening,
Using lemma PRECONDITION_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of PRECONDITION_TCC4.
Q.E.D.

C.42.6 Cpp_Examples.POSTCONDITION_TCC1

Terse proof for POSTCONDITION_TCC1.
POSTCONDITION_TCC1:

{1} ∀ (s: State): Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
POSTCONDITION_TCC1.1:

{1} Cpp_Type?(int)

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of POSTCONDITION_TCC1.1.
POSTCONDITION_TCC1.2:

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of POSTCONDITION_TCC1.2.
Q.E.D.

C.42.7 Cpp_Examples.spec02_TCC1

Terse proof for `spec02_TCC1`.

`spec02_TCC1:`

{1} plain_memory?(pm) ⊃ range(int)(42)

Applying disjunctive simplification to flatten sequent,
Expanding the definition of range,
Expanding the definition of range_integral,
Rewriting using int_binary, matching in *,
Using lemma min_int,
Using lemma max_int,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec02_TCC1`.
Q.E.D.

C.42.8 Cpp_Examples.spec02

Terse proof for `spec02`.

`spec02:`

{1} plain_memory?(pm) ⊃
valid(PRECONDITION, e2s(assign(pm, dt(int))(id(*n*), literal(42))), POSTCONDITION)

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpressions.== not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.- Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference Aligned_Ptr
Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_s pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_si pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s
Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt

continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while do_while_unroll for_as_while for_unroll

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r n2b a2b

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec02`.

Q.E.D.

C.42.9 Cpp_Examples.spec03

Terse proof for `spec03`.

`spec03:`

{1} plain_memory?(pm) ⊃
Valid(PRECONDITION, e2s(assign(pm, dt(int))(id(*n*), literal(42))), POSTCONDITION)

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpressions.== not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.- Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference Aligned_Ptr

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while do_while_unroll for_as_while for_unroll

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-

sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r n2b a2b

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec03`.

Q.E.D.

C.42.10 Cpp_Examples.spec04_TCC1

Terse proof for `spec04_TCC1`.

`spec04_TCC1:`

```
{1}  plain_memory?(pm) ⊃ range(char)(42)
```

Applying disjunctive simplification to flatten sequent,

Expanding the definition of range,

Expanding the definition of range_integral,

Rewriting using char_binary, matching in *,

Using lemma min_char,

Using lemma max_char,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec04_TCC1`.

Q.E.D.

C.42.11 Cpp_Examples.spec04_TCC2

Terse proof for `spec04_TCC2`.

`spec04_TCC2:`

```
{1}  plain_memory?(pm) ⊃ Cpp_Type?(char) ∧ cv(non_bool_integral_enum?)(char)
```

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Applying propositional simplification,

we get 2 subgoals:

`spec04_TCC2.1:`

```
{1}  Cpp_Type?(char)
```

Expanding the definition of Cpp_Type?,

Expanding the definition of subterm,

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec04_TCC2.1`.

`spec04_TCC2.2`:

```
{1}  cv(non_bool_integral_enum?)(char)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `spec04_TCC2.2`.

Q.E.D.

C.42.12 Cpp_Examples.spec04

Terse proof for `spec04`.

`spec04`:

```
{1}  plain_memory?(pm) ⊃
       valid(PRECONDITION,
             e2s(assign(pm, dt(char))(id(a), literal(42))) ##
              e2s(assign(pm, dt(char))(id(b), literal(42))),
             λ (s: State):
               OK?(read_data(pm, dt(char))(a)(s)) ∧
                 data(read_data(pm, dt(char))(a)(s)) = 42)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read

plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec04`.
Q.E.D.

C.42.13 Cpp_Examples.spec05

Terse proof for `spec05`.
`spec05:`

```
{1}  plain_memory?(pm) ⊃
       valid(PRECONDITION,
             e2s(assign(pm, dt_bool)(id(c), literal(TRUE))) ##
              if_else(l2r(pm, dt_bool)(id(c)),
                      e2s(assign(pm, dt(int))(id(n), literal(42))), skip),
             POSTCONDITION)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields
Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma
Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt
Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue

switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec05`.

Q.E.D.

C.42.14 Cpp_Examples.spec06_TCC1

Terse proof for `spec06_TCC1`.

`spec06_TCC1:`

```
{1}  plain_memory?(pm) ⊃ range(char)(0)
```

Applying disjunctive simplification to flatten sequent,

Expanding the definition of range,

Expanding the definition of range_integral,

Rewriting using char_binary, matching in *,
Using lemma min_char,
Using lemma max_char,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec06_TCC1`.
Q.E.D.

C.42.15 Cpp_Examples.spec06_TCC2

Terse proof for `spec06_TCC2`.
`spec06_TCC2:`

{1} plain_memory?(pm) ⊃ range(char)(1)

Applying disjunctive simplification to flatten sequent,
Expanding the definition of range,
Expanding the definition of range_integral,
Rewriting using char_binary, matching in *,
Using lemma min_char,
Using lemma max_char,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec06_TCC2`.
Q.E.D.

C.42.16 Cpp_Examples.spec06_TCC3

Terse proof for `spec06_TCC3`.
`spec06_TCC3:`

{1} plain_memory?(pm) ⊃
every[real](λ (x: real): rational_pred(x) ∧ integer_pred(x))((:1:))

Repeatedly Skolemizing and flattening,
Using lemma spec04_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `spec06_TCC3`.
Q.E.D.

C.42.17 Cpp_Examples.spec06_TCC4

Terse proof for `spec06_TCC4`.
`spec06_TCC4:`

{1} plain_memory?(pm) ⊃ range(int)(0)

Applying disjunctive simplification to flatten sequent,
Expanding the definition of range,
Expanding the definition of range_integral,
Rewriting using int_binary, matching in *,
Using lemma min_int,

Using lemma max_int,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec06_TCC4`.
Q.E.D.

C.42.18 Cpp_Examples.spec06

Terse proof for `spec06`.
`spec06:`

```
{1}  plain_memory?(pm) ⊃
       valid(PRECONDITION,
             e2s(assign(pm, dt(char))(id(a), literal(0))) ##
              e2s(assign(pm, dt(char))(id(b), literal(1)))
              ##
              switch(l2r(pm, dt(char))(id(a)), (:0, 1:),
                     case(0) ## skip ## case(1) ##
                      e2s(assign(pm, dt(int))(id(n), literal(42)))
                      ## break
                      ## default
                      ## e2s(assign(pm, dt(int))(id(n), literal(0)))),
             POSTCONDITION)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields
Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma
Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt
Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue

break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec06`.

Q.E.D.

C.42.19 Cpp_Examples.spec07

Terse proof for `spec07`.

spec07:

{1} valid(PRECONDITION, while(literal(TRUE), skip), λ (s: State): FALSE)

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: literal id list_props.member postinc postdec not_exp preint predec IntExpressions.* IntExpressions.div IntExpressions.rem IntExpressions.- IntExpressions.■ IntExpressions.■ IntExpressions.< IntExpressions.> IntExpressions.<= IntExpressions.>= EqualityExpressions.== not_equal and_exp or_exp assign assign_times IntExpressions.+ assign_plus assign_minus assign_div assign_rem comma assign_bsr assign_bsl deref IntExpressions2.+ IntExpressions2.- Cpp_pointer.+ Cpp_pointer.- ptr_addr_ptr_of pointer_add_n_pointer_minus pointer_add_difference Aligned_Ptr

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_as_while do_while_unroll for_as_while for_unroll

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt

composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r n2b a2b

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of hang_result,

which is trivially true.

This completes the proof of `spec07`.

Q.E.D.

C.42.20 Cpp_Examples.N_TCC1

Terse proof for `N_TCC1`.

`N_TCC1:`

$$\{1\} \quad 2 \geq 0 \wedge \text{range(int)}(2)$$

Installing automatic rewrites from: int_binary

Trying repeated skolemization, instantiation, and if-lifting,

we get 2 subgoals:

`N_TCC1.1:`

$$\{1\} \quad 2 \leq \text{max(range\_int)}$$

Using lemma max_int,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `N_TCC1.1`.

`N_TCC1.2:`

$$\{1\} \quad \text{min(range\_int)} \leq 2$$

Using lemma min_int,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `N_TCC1.2`.

Q.E.D.

C.42.21 Cpp_Examples.range_int_values_TCC1

Terse proof for `range_int_values_TCC1`.

`range_int_values_TCC1:`

{1} $\forall$ (i: int):
$-32767 \leq i \wedge i \leq 32767 \supset$ Cpp_Type?(int) $\wedge$ cv(non_bool_integral_enum?)(int)

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`range_int_values_TCC1.1:`

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_int_values_TCC1.1`.
`range_int_values_TCC1.2:`

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_int_values_TCC1.2`.
Q.E.D.

C.42.22 Cpp_Examples.range_int_values

Terse proof for `range_int_values`.
`range_int_values:`

{1} $\forall$ (i: int): $-32767 \leq i \wedge i \leq 32767 \supset$ range(int)(i)

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Expanding the definition of range_integral,
Rewriting using int_binary, matching in *,
Using lemma max_int,
Using lemma min_int,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_int_values`.
Q.E.D.

C.42.23 Cpp_Examples.spec08_TCC1

Terse proof for `spec08_TCC1`.

`spec08_TCC1:`

```
{1}  plain_memory?(pm) ⊃ Cpp_Type?(int) ∧ v(non_bool_integral?)(int)
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`spec08_TCC1.1:`

```
{-1}  plain_memory?(pm)
{1}   Cpp_Type?(int)
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `spec08_TCC1.1`.

`spec08_TCC1.2:`

```
{-1}  plain_memory?(pm)
{1}   v(non_bool_integral?)(int)
```

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `spec08_TCC1.2`.
Q.E.D.

C.42.24 Cpp_Examples.spec08

Terse proof for `spec08`.

`spec08:`

```
{1}  plain_memory?(pm) ⊃
       valid(PRECONDITION,
             e2s(assign(pm, dt(int))(id(n), literal(0))) ##
              e2s(assign(pm, dt(int))(id(m), literal(0)))
              ##
              while(N + 1, lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
                    e2s(postinc(pm, int)(id(n))) ## e2s(postinc(pm, int)(id(m)))),
             λ (s: State):
               OK?(read_data(pm, dt(int))(m)(s)) ⊃
                 data(read_data(pm, dt(int))(m)(s)) = N)
```

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!

stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: N

Installing automatic rewrites from: range_int_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec08`.

Q.E.D.

C.42.25 Cpp_Examples.spec09

Terse proof for `spec09`.

`spec09:`

$$
\begin{array}{l}
\{1\}\quad \text{plain\_memory?(pm)} \supset \\
\quad\quad \text{valid(PRECONDITION,} \\
\quad\quad\quad\quad \text{e2s(assign(pm, dt(int))(id}(m)\text{), literal(0))) \#\#} \\
\quad\quad\quad\quad\ \text{for}(N+1,\ \text{e2s(assign(pm, dt(int))(id}(n)\text{), literal(0))),} \\
\quad\quad\quad\quad\quad\quad \text{lt(int)(l2r(pm, dt(int))(id}(n)\text{)), literal}(N)\text{),} \\
\quad\quad\quad\quad\quad\quad \text{postinc(pm, int)(id}(n)\text{)), e2s(postinc(pm, int)(id}(m)\text{))),} \\
\quad\quad\quad\quad \lambda\ (s\text{: State}): \\
\quad\quad\quad\quad\quad \text{OK?(read\_data(pm, dt(int))}(m)(s)) \supset \\
\quad\quad\quad\quad\quad\quad \text{data(read\_data(pm, dt(int))}(m)(s)) = N)
\end{array}
$$

Installing automatic rewrites from: valid Valid PRECONDITION POSTCONDITION

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_of_spec address_of_spec2 add_simplification add_spec add_spec2 same_array_spec same_array_spec2 mod_range size_of uidt check_bounds_spec er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr

postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok compo-

sition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: N

Installing automatic rewrites from: range_int_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `spec09`.

Q.E.D.

C.42.26 Cpp_Examples.spec10_TCC1

Terse proof for `spec10_TCC1`.

`spec10_TCC1:`

```
{1}   plain_memory?(pm) ⊃
        (∀ (s₀: State, res: StmtResult[State]):
           OK?(res) ⊃
             OK?[State](res) ∨
               Break?[State](res) ∨
                 Continue?[State](res) ∨
                   Return?[State](res) ∨
                     Switch?[State](res) ∨ Default?[State](res) ∨ Exception?[State](res))
```

Repeatedly Skolemizing and flattening,

This completes the proof of `spec10_TCC1`.

Q.E.D.

C.42.27 Cpp_Examples.spec10

Terse proof for `spec10`.

spec10:

{1} plain_memory?(pm) ⊃
valid(PRECONDITION,
for(e2s(assign(pm, dt(int))(id(n), literal(0))),
lt(int)(l2r(pm, dt(int))(id(n)), literal(N)),
postinc(pm, int)(id(n)), skip)
((λ (s_0: State):
λ (res: StmtResult[State]):
OK?(res) ∧
OK?(read_data(pm, dt(int))(n)(state(res))) ∧
0 ≤ data(read_data(pm, dt(int))(n)(state(res))) ∧
data(read_data(pm, dt(int))(n)(state(res))) < N),
(λ (s_0: State):
λ (res: StmtResult[State]):
OK?(res) ∧
OK?(read_data(pm, dt(int))(n)(state(res))) ∧
data(read_data(pm, dt(int))(n)(state(res))) = N),
(λ (s_0: State):
λ (res: StmtResult[State]):
OK?(res) ∧
OK?(read_data(pm, dt(int))(n)(state(res))) ∧
data(read_data(pm, dt(int))(n)(state(res))) ≤ N),
(λ (s_0: State):
λ (s: State):
IF OK?(read_data(pm, dt(int))(n)(s)) ∧
data(read_data(pm, dt(int))(n)(s)) ≤ N
THEN N − data(read_data(pm, dt(int))(n)(s))
ELSE N
ENDIF),
extend[[numfield, numfield], [int, int], bool, FALSE]
(λ (i, j: int): i < j)),
λ (s: State):
OK?(read_data(pm, dt(int))(n)(s)) ⊃
data(read_data(pm, dt(int))(n)(s)) = N)

This completes the proof of spec10.

C.43 Proofs for Cpp_Integral_Types (types.pvs)

C.43.1 Cpp_Integral_Types.range_integral_TCC1

Terse proof for range_integral_TCC1.

range_integral_TCC1:

{1} ∀ (typ: (non_bool_integral?)):
typ = uchar ⊃ ¬ empty?[int](extend[int, nat, bool, FALSE](range_uchar))

Repeatedly Skolemizing and flattening,
Adding type constraints for range_uchar,
Keeping (-4 1) and hiding *,

Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC1`.
Q.E.D.

C.43.2 Cpp_Integral_Types.range_integral_TCC2

Terse proof for `range_integral_TCC2`.

`range_integral_TCC2`:

```
{1}   ∀ (typ: (non_bool_integral?)):
        typ = ushort ⊃ ¬ empty?[int](extend[int, nat, bool, FALSE](range_ushort))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for range_ushort,
Keeping (-4 1) and hiding *,
Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC2`.
Q.E.D.

C.43.3 Cpp_Integral_Types.range_integral_TCC3

Terse proof for `range_integral_TCC3`.

`range_integral_TCC3`:

```
{1}   ∀ (typ: (non_bool_integral?)):
        typ = uint ⊃ ¬ empty?[int](extend[int, nat, bool, FALSE](range_uint))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for range_uint,
Keeping (-4 1) and hiding *,
Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC3`.
Q.E.D.

C.43.4 Cpp_Integral_Types.range_integral_TCC4

Terse proof for `range_integral_TCC4`.

`range_integral_TCC4`:

```
{1}   ∀ (typ: (non_bool_integral?)):
        typ = ulong ⊃ ¬ empty?[int](extend[int, nat, bool, FALSE](range_ulong))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for range_ulong,
Keeping (-4 1) and hiding *,
Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC4`.
Q.E.D.

C.43.5 Cpp_Integral_Types.range_integral_TCC5

Terse proof for `range_integral_TCC5`.
`range_integral_TCC5:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       typ = ulonglong ⊃ ¬ empty?[int](extend[int, nat, bool, FALSE](range_ulonglong))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for range_ulonglong,
Keeping (-4 1) and hiding *,
Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_TCC5`.
Q.E.D.

C.43.6 Cpp_Integral_Types.range_integral_TCC6

Terse proof for `range_integral_TCC6`.
`range_integral_TCC6:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       ¬ (bool?(typ) ∨
            float?(typ) ∨
              double?(typ) ∨
                longdouble?(typ) ∨
                  void?(typ) ∨
                    array?(typ) ∨
                      function?(typ) ∨
                        pointer?(typ) ∨
                          reference?(typ) ∨
                            class?(typ) ∨
                              union?(typ) ∨
                                bitfield?(typ) ∨
                                  enum?(typ) ∨ pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of non_bool_integral?,
Expanding the definition of signed_integer?,
Expanding the definition of unsigned_integer?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_integral_TCC6`.
Q.E.D.

C.43.7 Cpp_Integral_Types.range_floating_point_TCC1

Terse proof for `range_floating_point_TCC1`.

`range_floating_point_TCC1`:

```
{1}   ∀ (typ: (floating_point?)):
        ¬ (uchar?(typ) ∨
             schar?(typ) ∨
              char?(typ) ∨
               short?(typ) ∨
                int?(typ) ∨
                 long?(typ) ∨
                  longlong?(typ) ∨
                   ushort?(typ) ∨
                    uint?(typ) ∨
                     ulong?(typ) ∨
                      ulonglong?(typ) ∨
                       wchar_t?(typ) ∨
                        bool?(typ) ∨
                         void?(typ) ∨
                          array?(typ) ∨
                           function?(typ) ∨
                            pointer?(typ) ∨
                             reference?(typ) ∨
                              class?(typ) ∨
                               union?(typ) ∨
                                bitfield?(typ) ∨
                                 enum?(typ) ∨
                                  pointer_to_member?(typ) ∨ const?(typ) ∨ volatile?(typ))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of floating_point?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_floating_point_TCC1`.
Q.E.D.

C.43.8 Cpp_Integral_Types.dt_integral_TCC1

Terse proof for `dt_integral_TCC1`.

`dt_integral_TCC1`:

```
{1}   ∀ (typ: (non_bool_integral?)):
        typ = uchar ⊃
         (∀ (x: int): x ≥ 0 ∧ (range_uchar)(x) ≡ range_integral(typ)(x)) ∧
          (∀ (x: int): x ≥ 0 ∧ (range_uchar)(x) ≡ range_integral(typ)(x)) ∧
           (∀ (x1: [list[Byte], Address]):
                every[(range_uchar)]((range_integral(typ)))(dt_uchar`from_byte(x1)))
            ∧ pod_data_type?[(range_integral(typ))](dt_uchar)
```

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`dt_integral_TCC1.1:`

{-1} non_bool_integral?(uchar)
{-2} uchar?(typ′)

{1} pod_data_type?⌊((range_integral(typ′)))⌋(dt_uchar)

Adding type constraints for dt_uchar,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_integral_TCC1.1.1:`

{-1} uninterpreted_data_type?(dt_uchar`uidt)
{-2} ∀ (d: ((range_uchar)), a: Address):
valid?(uidt(dt_uchar))(to_byte(dt_uchar)(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_uchar))(l, a) ≡ up?(from_byte(dt_uchar)(l, a))
{-4} ∀ (d: ((range_uchar)), a: Address):
down(from_byte(dt_uchar)(to_byte(dt_uchar)(d, a), a)) = d
{-5} ∀ (l: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_uchar))(l, a_1) ≡ valid?(uidt(dt_uchar))(l, a_2)
{-6} size(uidt(dt_uchar)) > 0
{-7} non_bool_integral?(uchar)
{-8} uchar?(typ′)

{1} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_uchar))(to_byte(dt_uchar)(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC1.1.1`.

`dt_integral_TCC1.1.2`:

{-1}	uninterpreted_data_type?(dt_uchar‘uidt)
{-2}	∀ (d: ((range_uchar)), a: Address): valid?(uidt(dt_uchar))(to_byte(dt_uchar)(d, a), a)
{-3}	∀ (l: list[Byte], a: Address): valid?(uidt(dt_uchar))(l, a) ≡ up?(from_byte(dt_uchar)(l, a))
{-4}	∀ (d: ((range_uchar)), a: Address): down(from_byte(dt_uchar)(to_byte(dt_uchar)(d, a), a)) = d
{-5}	∀ (l: list[Byte], a_1, a_2: Address): valid?(uidt(dt_uchar))(l, a_1) ≡ valid?(uidt(dt_uchar))(l, a_2)
{-6}	size(uidt(dt_uchar)) > 0
{-7}	non_bool_integral?(uchar)
{-8}	uchar?(typ′)
{1}	∀ (d: ((range_integral(typ′))), a: Address): down(from_byte(dt_uchar)(to_byte(dt_uchar)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC1.1.2`.

`dt_integral_TCC1.2`:

{-1}	non_bool_integral?(uchar)
{-2}	uchar?(typ′)
{1}	∀ (x_1: [list[Byte], Address]): every[(range_uchar)]((range_integral(uchar)))(dt_uchar‘from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of `dt_integral_TCC1.2`.

`dt_integral_TCC1.3`:

{-1}	non_bool_integral?(uchar)
{-2}	uchar?(typ′)
{1}	∀ (x: int): $x \geq 0 \wedge$ (range_uchar)(x) ≡ IF $x \geq 0$ THEN range_uchar(x) ELSE FALSE ENDIF

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC1.3`.
Q.E.D.

C.43.9 Cpp_Integral_Types.dt_integral_TCC2

Terse proof for `dt_integral_TCC2`.

`dt_integral_TCC2:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       typ = schar ⊃
        (∀ (x: int): (range_schar)(x) ≡ range_integral(typ)(x)) ∧
         (∀ (x: int): (range_schar)(x) ≡ range_integral(typ)(x)) ∧
          (∀ (x1: [list[Byte], Address]):
               every[(range_schar)]((range_integral(typ)))(dt_schar`from_byte(x1)))
           ∧ pod_data_type?[(range_integral(typ))](dt_schar)
```

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_integral_TCC2.1:`

```
{-1}  non_bool_integral?(schar)
{-2}  schar?(typ′)
------------------------------------------------------------
{1}   pod_data_type?[((range_integral(typ′)))](dt_schar)
```

Adding type constraints for dt_schar,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_integral_TCC2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_schar`uidt)
{-2}  ∀ (d: ((range_schar)), a: Address):
        valid?(uidt(dt_schar))(to_byte(dt_schar)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_schar))(l, a) ≡ up?(from_byte(dt_schar)(l, a))
{-4}  ∀ (d: ((range_schar)), a: Address):
        down(from_byte(dt_schar)(to_byte(dt_schar)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_schar))(l, a1) ≡ valid?(uidt(dt_schar))(l, a2)
{-6}  size(uidt(dt_schar)) > 0
{-7}  non_bool_integral?(schar)
{-8}  schar?(typ′)
------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        valid?(uidt(dt_schar))(to_byte(dt_schar)(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

which is trivially true.

This completes the proof of `dt_integral_TCC2.1.1`.

`dt_integral_TCC2.1.2:`

```
{-1}  uninterpreted_data_type?(dt_schar`uidt)
{-2}  ∀ (d: ((range_schar)), a: Address):
        valid?(uidt(dt_schar))(to_byte(dt_schar)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_schar))(l, a) ≡ up?(from_byte(dt_schar)(l, a))
{-4}  ∀ (d: ((range_schar)), a: Address):
        down(from_byte(dt_schar)(to_byte(dt_schar)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_schar))(l, a1) ≡ valid?(uidt(dt_schar))(l, a2)
{-6}  size(uidt(dt_schar)) > 0
{-7}  non_bool_integral?(schar)
{-8}  schar?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_schar)(to_byte(dt_schar)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC2.1.2`.
`dt_integral_TCC2.2:`

```
{-1}  non_bool_integral?(schar)
{-2}  schar?(typ′)
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_schar)]((range_integral(schar)))(dt_schar`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC2.2`.
Q.E.D.

C.43.10 Cpp_Integral_Types.dt_integral_TCC3

Terse proof for `dt_integral_TCC3`.
`dt_integral_TCC3:`

```
------------------------------------------------------------------------
{1}   ∀ (typ: (non_bool_integral?)):
        typ = char ⊃
          (∀ (x: int): (range_char)(x) ≡ range_integral(typ)(x)) ∧
            (∀ (x: int): (range_char)(x) ≡ range_integral(typ)(x)) ∧
              (∀ (x1: [list[Byte], Address]):
                 every[(range_char)]((range_integral(typ)))(dt_char`from_byte(x1)))
                ∧ pod_data_type?[(range_integral(typ))](dt_char)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:
`dt_integral_TCC3.1:`

```
{-1}  non_bool_integral?(char)
{-2}  char?(typ′)
------------------------------------------------------------------------
{1}   pod_data_type?[((range_integral(typ′)))](dt_char)
```

Adding type constraints for dt_char,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC3.1.1:`

```
{-1}  uninterpreted_data_type?(dt_char`uidt)
{-2}  ∀ (d: ((range_char)), a: Address):
        valid?(uidt(dt_char))(to_byte(dt_char)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_char))(l, a) ≡ up?(from_byte(dt_char)(l, a))
{-4}  ∀ (d: ((range_char)), a: Address):
        down(from_byte(dt_char)(to_byte(dt_char)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_char))(l, a1) ≡ valid?(uidt(dt_char))(l, a2)
{-6}  size(uidt(dt_char)) > 0
{-7}  non_bool_integral?(char)
{-8}  char?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        valid?(uidt(dt_char))(to_byte(dt_char)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC3.1.1`.
`dt_integral_TCC3.1.2:`

```
{-1}  uninterpreted_data_type?(dt_char`uidt)
{-2}  ∀ (d: ((range_char)), a: Address):
        valid?(uidt(dt_char))(to_byte(dt_char)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_char))(l, a) ≡ up?(from_byte(dt_char)(l, a))
{-4}  ∀ (d: ((range_char)), a: Address):
        down(from_byte(dt_char)(to_byte(dt_char)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_char))(l, a1) ≡ valid?(uidt(dt_char))(l, a2)
{-6}  size(uidt(dt_char)) > 0
{-7}  non_bool_integral?(char)
{-8}  char?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_char)(to_byte(dt_char)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,

which is trivially true.

This completes the proof of `dt_integral_TCC3.1.2`.

`dt_integral_TCC3.2`:

```
{-1}  non_bool_integral?(char)
{-2}  char?(typ′)
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_char)]((range_integral(char)))(dt_char‘from_byte(x1))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC3.2`.

Q.E.D.

C.43.11 Cpp_Integral_Types.dt_integral_TCC4

Terse proof for `dt_integral_TCC4`.

`dt_integral_TCC4`:

```
------------------------------------------------------------------------
{1}   ∀ (typ: (non_bool_integral?)):
        typ = short ⊃
         (∀ (x: int): (range_short)(x) ≡ range_integral(typ)(x)) ∧
          (∀ (x: int): (range_short)(x) ≡ range_integral(typ)(x)) ∧
           (∀ (x1: [list[Byte], Address]):
              every[(range_short)]((range_integral(typ)))(dt_short‘from_byte(x1)))
            ∧ pod_data_type?[(range_integral(typ))](dt_short)
```

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_integral_TCC4.1`:

```
{-1}  non_bool_integral?(short)
{-2}  short?(typ′)
------------------------------------------------------------------------
{1}   pod_data_type?[((range_integral(typ′)))](dt_short)
```

Adding type constraints for dt_short,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_integral_TCC4.1.1:`

```
{-1}  uninterpreted_data_type?(dt_short`uidt)
{-2}  ∀ (d: ((range_short)), a: Address):
        valid?(uidt(dt_short))(to_byte(dt_short)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_short))(l, a) ≡ up?(from_byte(dt_short)(l, a))
{-4}  ∀ (d: ((range_short)), a: Address):
        down(from_byte(dt_short)(to_byte(dt_short)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_short))(l, a1) ≡ valid?(uidt(dt_short))(l, a2)
{-6}  size(uidt(dt_short)) > 0
{-7}  non_bool_integral?(short)
{-8}  short?(typ′)
-------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        valid?(uidt(dt_short))(to_byte(dt_short)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC4.1.1`.
`dt_integral_TCC4.1.2:`

```
{-1}  uninterpreted_data_type?(dt_short`uidt)
{-2}  ∀ (d: ((range_short)), a: Address):
        valid?(uidt(dt_short))(to_byte(dt_short)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_short))(l, a) ≡ up?(from_byte(dt_short)(l, a))
{-4}  ∀ (d: ((range_short)), a: Address):
        down(from_byte(dt_short)(to_byte(dt_short)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_short))(l, a1) ≡ valid?(uidt(dt_short))(l, a2)
{-6}  size(uidt(dt_short)) > 0
{-7}  non_bool_integral?(short)
{-8}  short?(typ′)
-------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_short)(to_byte(dt_short)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC4.1.2`.
`dt_integral_TCC4.2:`

```
{-1}  non_bool_integral?(short)
{-2}  short?(typ′)
-------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_short)]((range_integral(short)))(dt_short`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC4.2`.
Q.E.D.

C.43.12 Cpp_Integral_Types.dt_integral_TCC5

Terse proof for `dt_integral_TCC5`.
`dt_integral_TCC5:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       typ = int ⊃
        (∀ (x: int): (range_int)(x) ≡ range_integral(typ)(x)) ∧
         (∀ (x: int): (range_int)(x) ≡ range_integral(typ)(x)) ∧
          (∀ (x_1: [list[Byte], Address]):
               every[(range_int)]((range_integral(typ)))(dt_int`from_byte(x_1)))
            ∧ pod_data_type?[(range_integral(typ))](dt_int)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_integral_TCC5.1:`

```
{-1}  non_bool_integral?(int)
{-2}  int?(typ′)
------------------------------------------------------------
{1}   pod_data_type?[((range_integral(typ′)))](dt_int)
```

Adding type constraints for dt_int,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC5.1.1:`

```
{-1}  uninterpreted_data_type?(dt_int`uidt)
{-2}  ∀ (d: ((range_int)), a: Address):
        valid?(uidt(dt_int))(to_byte(dt_int)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_int))(l, a) ≡ up?(from_byte(dt_int)(l, a))
{-4}  ∀ (d: ((range_int)), a: Address):
        down(from_byte(dt_int)(to_byte(dt_int)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(dt_int))(l, a_1) ≡ valid?(uidt(dt_int))(l, a_2)
{-6}  size(uidt(dt_int)) > 0
{-7}  non_bool_integral?(int)
{-8}  int?(typ′)
------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        valid?(uidt(dt_int))(to_byte(dt_int)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.

This completes the proof of `dt_integral_TCC5.1.1`.
`dt_integral_TCC5.1.2`:

```
{-1}  uninterpreted_data_type?(dt_int`uidt)
{-2}  ∀ (d: ((range_int)), a: Address):
        valid?(uidt(dt_int))(to_byte(dt_int)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_int))(l, a) ≡ up?(from_byte(dt_int)(l, a))
{-4}  ∀ (d: ((range_int)), a: Address):
        down(from_byte(dt_int)(to_byte(dt_int)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(dt_int))(l, a_1) ≡ valid?(uidt(dt_int))(l, a_2)
{-6}  size(uidt(dt_int)) > 0
{-7}  non_bool_integral?(int)
{-8}  int?(typ′)
-------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_int)(to_byte(dt_int)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC5.1.2`.
`dt_integral_TCC5.2`:

```
{-1}  non_bool_integral?(int)
{-2}  int?(typ′)
-------------------------------------------------------------------
{1}   ∀ (x_1: [list[Byte], Address]):
        every[(range_int)]((range_integral(int)))(dt_int`from_byte(x_1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC5.2`.
Q.E.D.

C.43.13 Cpp_Integral_Types.dt_integral_TCC6

Terse proof for `dt_integral_TCC6`.
`dt_integral_TCC6`:

```
-------------------------------------------------------------------
{1}   ∀ (typ: (non_bool_integral?)):
        typ = long ⊃
          (∀ (x: int): (range_long)(x) ≡ range_integral(typ)(x)) ∧
            (∀ (x: int): (range_long)(x) ≡ range_integral(typ)(x)) ∧
              (∀ (x_1: [list[Byte], Address]):
                 every[(range_long)]((range_integral(typ)))(dt_long`from_byte(x_1)))
                ∧ pod_data_type?[(range_integral(typ))](dt_long)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_integral_TCC6.1:`

```
{-1}  non_bool_integral?(long)
{-2}  long?(typ′)
------------------------------------------------------------------------
{1}   pod_data_type?⌊((range_integral(typ′)))⌋(dt_long)
```

Adding type constraints for dt_long,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC6.1.1:`

```
{-1}  uninterpreted_data_type?(dt_long`uidt)
{-2}  ∀ (d: ((range_long)), a: Address):
        valid?(uidt(dt_long))(to_byte(dt_long)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_long))(l, a) ≡ up?(from_byte(dt_long)(l, a))
{-4}  ∀ (d: ((range_long)), a: Address):
        down(from_byte(dt_long)(to_byte(dt_long)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(dt_long))(l, a_1) ≡ valid?(uidt(dt_long))(l, a_2)
{-6}  size(uidt(dt_long)) > 0
{-7}  non_bool_integral?(long)
{-8}  long?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        valid?(uidt(dt_long))(to_byte(dt_long)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC6.1.1`.
`dt_integral_TCC6.1.2:`

```
{-1}  uninterpreted_data_type?(dt_long`uidt)
{-2}  ∀ (d: ((range_long)), a: Address):
        valid?(uidt(dt_long))(to_byte(dt_long)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_long))(l, a) ≡ up?(from_byte(dt_long)(l, a))
{-4}  ∀ (d: ((range_long)), a: Address):
        down(from_byte(dt_long)(to_byte(dt_long)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(dt_long))(l, a_1) ≡ valid?(uidt(dt_long))(l, a_2)
{-6}  size(uidt(dt_long)) > 0
{-7}  non_bool_integral?(long)
{-8}  long?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_long)(to_byte(dt_long)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC6.1.2`.
`dt_integral_TCC6.2:`

{-1} non_bool_integral?(long)
{-2} long?(typ′)

{1} $\forall\ (x_1\colon\ \lfloor\text{list}\lfloor\text{Byte}\rfloor,\ \text{Address}\rfloor)\colon$
$\text{every}\lfloor(\text{range\_long})\rfloor((\text{range\_integral}(\text{long})))(\text{dt\_long`from\_byte}(x_1))$

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC6.2`.
Q.E.D.

C.43.14 Cpp_Integral_Types.dt_integral_TCC7

Terse proof for `dt_integral_TCC7`.
`dt_integral_TCC7:`

{1} $\forall\ (\text{typ}\colon\ (\text{non\_bool\_integral?}))\colon$
$\text{typ} = \text{longlong} \supset$
$(\forall\ (x\colon\ \text{int})\colon\ (\text{range\_longlong})(x) \equiv \text{range\_integral}(\text{typ})(x)) \wedge$
$(\forall\ (x\colon\ \text{int})\colon\ (\text{range\_longlong})(x) \equiv \text{range\_integral}(\text{typ})(x)) \wedge$
$(\forall\ (x_1\colon\ \lfloor\text{list}\lfloor\text{Byte}\rfloor,\ \text{Address}\rfloor)\colon$
$\text{every}\lfloor(\text{range\_longlong})\rfloor((\text{range\_integral}(\text{typ})))(\text{dt\_longlong`from\_byte}(x_1)))$
$\wedge\ \text{pod\_data\_type?}\lfloor(\text{range\_integral}(\text{typ}))\rfloor(\text{dt\_longlong})$

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_integral_TCC7.1:`

{-1} non_bool_integral?(longlong)
{-2} longlong?(typ′)

{1} $\text{pod\_data\_type?}\lfloor((\text{range\_integral}(\text{typ}')))\rfloor(\text{dt\_longlong})$

Adding type constraints for dt_longlong,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_integral_TCC7.1.1:`

```
{-1}  uninterpreted_data_type?(dt_longlong`uidt)
{-2}  ∀ (d: ((range_longlong)), a: Address):
        valid?(uidt(dt_longlong))(to_byte(dt_longlong)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_longlong))(l, a) ≡ up?(from_byte(dt_longlong)(l, a))
{-4}  ∀ (d: ((range_longlong)), a: Address):
        down(from_byte(dt_longlong)(to_byte(dt_longlong)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_longlong))(l, a1) ≡ valid?(uidt(dt_longlong))(l, a2)
{-6}  size(uidt(dt_longlong)) > 0
{-7}  non_bool_integral?(longlong)
{-8}  longlong?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        valid?(uidt(dt_longlong))(to_byte(dt_longlong)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC7.1.1`.
`dt_integral_TCC7.1.2:`

```
{-1}  uninterpreted_data_type?(dt_longlong`uidt)
{-2}  ∀ (d: ((range_longlong)), a: Address):
        valid?(uidt(dt_longlong))(to_byte(dt_longlong)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_longlong))(l, a) ≡ up?(from_byte(dt_longlong)(l, a))
{-4}  ∀ (d: ((range_longlong)), a: Address):
        down(from_byte(dt_longlong)(to_byte(dt_longlong)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_longlong))(l, a1) ≡ valid?(uidt(dt_longlong))(l, a2)
{-6}  size(uidt(dt_longlong)) > 0
{-7}  non_bool_integral?(longlong)
{-8}  longlong?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_longlong)(to_byte(dt_longlong)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC7.1.2`.
`dt_integral_TCC7.2:`

```
{-1}  non_bool_integral?(longlong)
{-2}  longlong?(typ′)
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_longlong)]((range_integral(longlong)))(dt_longlong`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_integral_TCC7.2`.
Q.E.D.

C.43.15 Cpp_Integral_Types.dt_integral_TCC8

Terse proof for `dt_integral_TCC8`.
`dt_integral_TCC8:`

{1} ∀ (typ: (non_bool_integral?)):
typ = ushort ⊃
(∀ (x: int): $x \geq 0$ ∧ (range_ushort)(x) ≡ range_integral(typ)(x)) ∧
(∀ (x: int): $x \geq 0$ ∧ (range_ushort)(x) ≡ range_integral(typ)(x)) ∧
(∀ (x_1: [list[Byte], Address]):
every[(range_ushort)]((range_integral(typ)))(dt_ushort`from_byte(x_1)))
∧ pod_data_type?[(range_integral(typ))](dt_ushort)

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_integral_TCC8.1:`

{-1} non_bool_integral?(ushort)
{-2} ushort?(typ′)

{1} pod_data_type?[((range_integral(typ′)))](dt_ushort)

Adding type constraints for dt_ushort,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC8.1.1:`

{-1} uninterpreted_data_type?(dt_ushort`uidt)
{-2} ∀ (d: ((range_ushort)), a: Address):
valid?(uidt(dt_ushort))(to_byte(dt_ushort)(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_ushort))(l, a) ≡ up?(from_byte(dt_ushort)(l, a))
{-4} ∀ (d: ((range_ushort)), a: Address):
down(from_byte(dt_ushort)(to_byte(dt_ushort)(d, a), a)) = d
{-5} ∀ (l: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_ushort))(l, a_1) ≡ valid?(uidt(dt_ushort))(l, a_2)
{-6} size(uidt(dt_ushort)) > 0
{-7} non_bool_integral?(ushort)
{-8} ushort?(typ′)

{1} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_ushort))(to_byte(dt_ushort)(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,

Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC8.1.1`.
`dt_integral_TCC8.1.2`:

```
{-1}  uninterpreted_data_type?(dt_ushort`uidt)
{-2}  ∀ (d: ((range_ushort)), a: Address):
        valid?(uidt(dt_ushort))(to_byte(dt_ushort)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_ushort))(l, a) ≡ up?(from_byte(dt_ushort)(l, a))
{-4}  ∀ (d: ((range_ushort)), a: Address):
        down(from_byte(dt_ushort)(to_byte(dt_ushort)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_ushort))(l, a1) ≡ valid?(uidt(dt_ushort))(l, a2)
{-6}  size(uidt(dt_ushort)) > 0
{-7}  non_bool_integral?(ushort)
{-8}  ushort?(typ′)
-------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_ushort)(to_byte(dt_ushort)(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC8.1.2`.
`dt_integral_TCC8.2`:

```
{-1}  non_bool_integral?(ushort)
{-2}  ushort?(typ′)
-------------------------------------------------------------------------
{1}   ∀ (x1: ⌊list[Byte], Address⌋):
        every[(range_ushort)]((range_integral(ushort)))(dt_ushort`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of `dt_integral_TCC8.2`.
`dt_integral_TCC8.3`:

```
{-1}  non_bool_integral?(ushort)
{-2}  ushort?(typ′)
-------------------------------------------------------------------------
{1}   ∀ (x: int):
        x ≥ 0 ∧ (range_ushort)(x) ≡ IF x ≥ 0 THEN range_ushort(x) ELSE FALSE ENDIF
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC8.3`.
Q.E.D.

C.43.16 Cpp_Integral_Types.dt_integral_TCC9

Terse proof for `dt_integral_TCC9`.

`dt_integral_TCC9:`

{1} ∀ (typ: (non_bool_integral?)):
typ = uint ⊃
(∀ (x: int): x ≥ 0 ∧ (range_uint)(x) ≡ range_integral(typ)(x)) ∧
(∀ (x: int): x ≥ 0 ∧ (range_uint)(x) ≡ range_integral(typ)(x)) ∧
(∀ (x₁: [list[Byte], Address]):
every[(range_uint)]((range_integral(typ)))(dt_uint`from_byte(x₁)))
∧ pod_data_type?[(range_integral(typ))](dt_uint)

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_integral_TCC9.1:`

{-1} non_bool_integral?(uint)
{-2} uint?(typ′)

{1} pod_data_type?[((range_integral(typ′)))](dt_uint)

Adding type constraints for dt_uint,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC9.1.1:`

{-1} uninterpreted_data_type?(dt_uint`uidt)
{-2} ∀ (d: ((range_uint)), a: Address):
valid?(uidt(dt_uint))(to_byte(dt_uint)(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_uint))(l, a) ≡ up?(from_byte(dt_uint)(l, a))
{-4} ∀ (d: ((range_uint)), a: Address):
down(from_byte(dt_uint)(to_byte(dt_uint)(d, a), a)) = d
{-5} ∀ (l: list[Byte], a₁, a₂: Address):
valid?(uidt(dt_uint))(l, a₁) ≡ valid?(uidt(dt_uint))(l, a₂)
{-6} size(uidt(dt_uint)) > 0
{-7} non_bool_integral?(uint)
{-8} uint?(typ′)

{1} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_uint))(to_byte(dt_uint)(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC9.1.1`.

`dt_integral_TCC9.1.2`:

{-1}	uninterpreted_data_type?(dt_uint‘uidt)
{-2}	∀ (d: ((range_uint)), a: Address): valid?(uidt(dt_uint))(to_byte(dt_uint)(d, a), a)
{-3}	∀ (l: list[Byte], a: Address): valid?(uidt(dt_uint))(l, a) ≡ up?(from_byte(dt_uint)(l, a))
{-4}	∀ (d: ((range_uint)), a: Address): down(from_byte(dt_uint)(to_byte(dt_uint)(d, a), a)) = d
{-5}	∀ (l: list[Byte], a_1, a_2: Address): valid?(uidt(dt_uint))(l, a_1) ≡ valid?(uidt(dt_uint))(l, a_2)
{-6}	size(uidt(dt_uint)) > 0
{-7}	non_bool_integral?(uint)
{-8}	uint?(typ′)
{1}	∀ (d: ((range_integral(typ′))), a: Address): down(from_byte(dt_uint)(to_byte(dt_uint)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC9.1.2`.
`dt_integral_TCC9.2`:

{-1}	non_bool_integral?(uint)
{-2}	uint?(typ′)
{1}	∀ (x_1: [list[Byte], Address]): every[(range_uint)]((range_integral(uint)))(dt_uint‘from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of `dt_integral_TCC9.2`.
`dt_integral_TCC9.3`:

{-1}	non_bool_integral?(uint)
{-2}	uint?(typ′)
{1}	∀ (x: int): $x \geq 0 \wedge$ (range_uint)(x) ≡ IF $x \geq 0$ THEN range_uint(x) ELSE FALSE ENDIF

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC9.3`.
Q.E.D.

C.43.17 Cpp_Integral_Types.dt_integral_TCC10

Terse proof for `dt_integral_TCC10`.

`dt_integral_TCC10:`

{1} ∀ (typ: (non_bool_integral?)):
typ = ulong ⊃
(∀ (x: int): x ≥ 0 ∧ (range_ulong)(x) ≡ range_integral(typ)(x)) ∧
(∀ (x: int): x ≥ 0 ∧ (range_ulong)(x) ≡ range_integral(typ)(x)) ∧
(∀ (x_1: [list[Byte], Address]):
every[(range_ulong)]((range_integral(typ)))(dt_ulong`from_byte(x_1)))
∧ pod_data_type?[(range_integral(typ))](dt_ulong)

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_integral_TCC10.1:`

{-1} non_bool_integral?(ulong)
{-2} ulong?(typ′)
{1} pod_data_type?[((range_integral(typ′)))](dt_ulong)

Adding type constraints for dt_ulong,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC10.1.1:`

{-1} uninterpreted_data_type?(dt_ulong`uidt)
{-2} ∀ (d: ((range_ulong)), a: Address):
valid?(uidt(dt_ulong))(to_byte(dt_ulong)(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_ulong))(l, a) ≡ up?(from_byte(dt_ulong)(l, a))
{-4} ∀ (d: ((range_ulong)), a: Address):
down(from_byte(dt_ulong)(to_byte(dt_ulong)(d, a), a)) = d
{-5} ∀ (l: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_ulong))(l, a_1) ≡ valid?(uidt(dt_ulong))(l, a_2)
{-6} size(uidt(dt_ulong)) > 0
{-7} non_bool_integral?(ulong)
{-8} ulong?(typ′)
{1} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_ulong))(to_byte(dt_ulong)(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC10.1.1`.

`dt_integral_TCC10.1.2`:

```
{-1}  uninterpreted_data_type?(dt_ulong`uidt)
{-2}  ∀ (d: ((range_ulong)), a: Address):
        valid?(uidt(dt_ulong))(to_byte(dt_ulong)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_ulong))(l, a) ≡ up?(from_byte(dt_ulong)(l, a))
{-4}  ∀ (d: ((range_ulong)), a: Address):
        down(from_byte(dt_ulong)(to_byte(dt_ulong)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_ulong))(l, a1) ≡ valid?(uidt(dt_ulong))(l, a2)
{-6}  size(uidt(dt_ulong)) > 0
{-7}  non_bool_integral?(ulong)
{-8}  ulong?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_ulong)(to_byte(dt_ulong)(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC10.1.2`.

`dt_integral_TCC10.2`:

```
{-1}  non_bool_integral?(ulong)
{-2}  ulong?(typ′)
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_ulong)]((range_integral(ulong)))(dt_ulong`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of `dt_integral_TCC10.2`.

`dt_integral_TCC10.3`:

```
{-1}  non_bool_integral?(ulong)
{-2}  ulong?(typ′)
------------------------------------------------------------------------
{1}   ∀ (x: int):
        x ≥ 0 ∧ (range_ulong)(x) ≡ IF x ≥ 0 THEN range_ulong(x) ELSE FALSE ENDIF
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC10.3`.
Q.E.D.

C.43.18 Cpp_Integral_Types.dt_integral_TCC11

Terse proof for `dt_integral_TCC11`.

`dt_integral_TCC11:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       typ = ulonglong ⊃
        (∀ (x: int): x ≥ 0 ∧ (range_ulonglong)(x) ≡ range_integral(typ)(x)) ∧
         (∀ (x: int): x ≥ 0 ∧ (range_ulonglong)(x) ≡ range_integral(typ)(x)) ∧
          (∀ (x1: [list[Byte], Address]):
               every[(range_ulonglong)]
                    ((range_integral(typ)))(dt_ulonglong`from_byte(x1)))
            ∧ pod_data_type?[(range_integral(typ))](dt_ulonglong)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_integral_TCC11.1:`

```
{-1}  non_bool_integral?(ulonglong)
{-2}  ulonglong?(typ')
------------------------------------------------------------
 {1}  pod_data_type?[((range_integral(typ')))](dt_ulonglong)
```

Adding type constraints for dt_ulonglong,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_integral_TCC11.1.1:`

```
{-1}  uninterpreted_data_type?(dt_ulonglong`uidt)
{-2}  ∀ (d: ((range_ulonglong)), a: Address):
        valid?(uidt(dt_ulonglong))(to_byte(dt_ulonglong)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_ulonglong))(l, a) ≡ up?(from_byte(dt_ulonglong)(l, a))
{-4}  ∀ (d: ((range_ulonglong)), a: Address):
        down(from_byte(dt_ulonglong)(to_byte(dt_ulonglong)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_ulonglong))(l, a1) ≡ valid?(uidt(dt_ulonglong))(l, a2)
{-6}  size(uidt(dt_ulonglong)) > 0
{-7}  non_bool_integral?(ulonglong)
{-8}  ulonglong?(typ')
------------------------------------------------------------
 {1}  ∀ (d: ((range_integral(typ'))), a: Address):
        valid?(uidt(dt_ulonglong))(to_byte(dt_ulonglong)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC11.1.1`.

`dt_integral_TCC11.1.2`:

{-1}	uninterpreted_data_type?(dt_ulonglong`uidt)
{-2}	∀ (d: ((range_ulonglong)), a: Address): valid?(uidt(dt_ulonglong))(to_byte(dt_ulonglong)(d, a), a)
{-3}	∀ (l: list[Byte], a: Address): valid?(uidt(dt_ulonglong))(l, a) ≡ up?(from_byte(dt_ulonglong)(l, a))
{-4}	∀ (d: ((range_ulonglong)), a: Address): down(from_byte(dt_ulonglong)(to_byte(dt_ulonglong)(d, a), a)) = d
{-5}	∀ (l: list[Byte], a_1, a_2: Address): valid?(uidt(dt_ulonglong))(l, a_1) ≡ valid?(uidt(dt_ulonglong))(l, a_2)
{-6}	size(uidt(dt_ulonglong)) > 0
{-7}	non_bool_integral?(ulonglong)
{-8}	ulonglong?(typ′)
{1}	∀ (d: ((range_integral(typ′))), a: Address): down(from_byte(dt_ulonglong)(to_byte(dt_ulonglong)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC11.1.2`.
`dt_integral_TCC11.2`:

{-1}	non_bool_integral?(ulonglong)
{-2}	ulonglong?(typ′)
{1}	∀ (x_1: [list[Byte], Address]): every[(range_ulonglong)]((range_integral(ulonglong)))(dt_ulonglong`from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of extend,
which is trivially true.
This completes the proof of `dt_integral_TCC11.2`.
`dt_integral_TCC11.3`:

{-1}	non_bool_integral?(ulonglong)
{-2}	ulonglong?(typ′)
{1}	∀ (x: int): $x \geq 0 \wedge$ (range_ulonglong)(x) ≡ IF $x \geq 0$ THEN range_ulonglong(x) ELSE FALSE ENDIF

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC11.3`.
Q.E.D.

C.43.19 Cpp_Integral_Types.dt_integral_TCC12

Terse proof for `dt_integral_TCC12`.

`dt_integral_TCC12:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       typ = wchar_t ⊃
        (∀ (x: int): (range_wchar_t)(x) ≡ range_integral(typ)(x)) ∧
         (∀ (x: int): (range_wchar_t)(x) ≡ range_integral(typ)(x)) ∧
          (∀ (x1: [list[Byte], Address]):
               every[(range_wchar_t)]((range_integral(typ)))(dt_wchar_t`from_byte(x1)))
           ∧ pod_data_type?[(range_integral(typ))](dt_wchar_t)
```

Repeatedly Skolemizing and flattening,

Replacing using formula -2,

Expanding the definition of range_integral,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_integral_TCC12.1:`

```
{-1}  non_bool_integral?(wchar_t)
{-2}  wchar_t?(typ')
------------------------------------------------------------
{1}   pod_data_type?[((range_integral(typ')))](dt_wchar_t)
```

Adding type constraints for dt_wchar_t,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_integral_TCC12.1.1:`

```
{-1}  uninterpreted_data_type?(dt_wchar_t`uidt)
{-2}  ∀ (d: ((range_wchar_t)), a: Address):
        valid?(uidt(dt_wchar_t))(to_byte(dt_wchar_t)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_wchar_t))(l, a) ≡ up?(from_byte(dt_wchar_t)(l, a))
{-4}  ∀ (d: ((range_wchar_t)), a: Address):
        down(from_byte(dt_wchar_t)(to_byte(dt_wchar_t)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_wchar_t))(l, a1) ≡ valid?(uidt(dt_wchar_t))(l, a2)
{-6}  size(uidt(dt_wchar_t)) > 0
{-7}  non_bool_integral?(wchar_t)
{-8}  wchar_t?(typ')
------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ'))), a: Address):
        valid?(uidt(dt_wchar_t))(to_byte(dt_wchar_t)(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_integral,

which is trivially true.

This completes the proof of `dt_integral_TCC12.1.1`.

`dt_integral_TCC12.1.2:`

```
{-1}  uninterpreted_data_type?(dt_wchar_t`uidt)
{-2}  ∀ (d: ((range_wchar_t)), a: Address):
        valid?(uidt(dt_wchar_t))(to_byte(dt_wchar_t)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_wchar_t))(l, a) ≡ up?(from_byte(dt_wchar_t)(l, a))
{-4}  ∀ (d: ((range_wchar_t)), a: Address):
        down(from_byte(dt_wchar_t)(to_byte(dt_wchar_t)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_wchar_t))(l, a1) ≡ valid?(uidt(dt_wchar_t))(l, a2)
{-6}  size(uidt(dt_wchar_t)) > 0
{-7}  non_bool_integral?(wchar_t)
{-8}  wchar_t?(typ′)
------------------------------------------------------------------------
{1}   ∀ (d: ((range_integral(typ′))), a: Address):
        down(from_byte(dt_wchar_t)(to_byte(dt_wchar_t)(d, a), a)) = d
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_integral,
which is trivially true.
This completes the proof of `dt_integral_TCC12.1.2`.
`dt_integral_TCC12.2:`

```
{-1}  non_bool_integral?(wchar_t)
{-2}  wchar_t?(typ′)
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_wchar_t)]((range_integral(wchar_t)))(dt_wchar_t`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of range_integral,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_integral_TCC12.2`.
Q.E.D.

C.43.20 Cpp_Integral_Types.dt_floating_point_TCC1

Terse proof for `dt_floating_point_TCC1`.
`dt_floating_point_TCC1:`

```
------------------------------------------------------------------------
{1}   ∀ (typ: (floating_point?)):
        typ = float ⊃
          (∀ (x: extended_real): (range_float)(x) ≡ range_floating_point(typ)(x)) ∧
            (∀ (x: extended_real): (range_float)(x) ≡ range_floating_point(typ)(x)) ∧
              (∀ (x1: [list[Byte], Address]):
                 every[(range_float)]((range_floating_point(typ)))(dt_float`from_byte(x1)))
                ∧ pod_data_type?[(range_floating_point(typ))](dt_float)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_floating_point,
Applying propositional simplification,

we get 2 subgoals:

`dt_floating_point_TCC1.1:`

```
{-1}  floating_point?(float)
{-2}  typ′ = float
------------------------------------------------------------------------
{1}   ∀ (x1: |list[Byte], Address|):
        every[(range_float)]((range_floating_point(float)))(dt_float‘from_byte(x1))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of range_floating_point,

which is trivially true.

This completes the proof of `dt_floating_point_TCC1.1`.

`dt_floating_point_TCC1.2:`

```
{-1}  floating_point?(float)
{-2}  typ′ = float
------------------------------------------------------------------------
{1}   pod_data_type?|((range_floating_point(typ′)))|(dt_float)
```

Adding type constraints for dt_float,

Expanding the definition of pod_data_type?,

Applying propositional simplification,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_floating_point_TCC1.2.1:`

```
{-1}  uninterpreted_data_type?(dt_float‘uidt)
{-2}  ∀ (d: ((range_float)), a: Address):
        valid?(uidt(dt_float))(to_byte(dt_float)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_float))(l, a) ≡ up?(from_byte(dt_float)(l, a))
{-4}  ∀ (d: ((range_float)), a: Address):
        down(from_byte(dt_float)(to_byte(dt_float)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_float))(l, a1) ≡ valid?(uidt(dt_float))(l, a2)
{-6}  size(uidt(dt_float)) > 0
{-7}  floating_point?(float)
{-8}  typ′ = float
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(typ′))), a: Address):
        valid?(uidt(dt_float))(to_byte(dt_float)(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range_floating_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_floating_point_TCC1.2.1`.

`dt_floating_point_TCC1.2.2:`

```
{-1}  uninterpreted_data_type?(dt_float`uidt)
{-2}  ∀ (d: ((range_float)), a: Address):
        valid?(uidt(dt_float))(to_byte(dt_float)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_float))(l, a) ≡ up?(from_byte(dt_float)(l, a))
{-4}  ∀ (d: ((range_float)), a: Address):
        down(from_byte(dt_float)(to_byte(dt_float)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_float))(l, a1) ≡ valid?(uidt(dt_float))(l, a2)
{-6}  size(uidt(dt_float)) > 0
{-7}  floating_point?(float)
{-8}  typ′ = float
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(typ′))), a: Address):
        down(from_byte(dt_float)(to_byte(dt_float)(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_floating_point,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_floating_point_TCC1.2.2`.
Q.E.D.

C.43.21 Cpp_Integral_Types.dt_floating_point_TCC2

Terse proof for `dt_floating_point_TCC2`.

`dt_floating_point_TCC2:`

```
------------------------------------------------------------------------
{1}   ∀ (typ: (floating_point?)):
        typ = double ⊃
          (∀ (x: extended_real): (range_double)(x) ≡ range_floating_point(typ)(x)) ∧
            (∀ (x: extended_real): (range_double)(x) ≡ range_floating_point(typ)(x)) ∧
              (∀ (x1: [list[Byte], Address]):
                 every[(range_double)]
                      ((range_floating_point(typ)))(dt_double`from_byte(x1)))
                ∧ pod_data_type?[(range_floating_point(typ))](dt_double)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_floating_point,
Applying propositional simplification,
we get 2 subgoals:

`dt_floating_point_TCC2.1:`

```
{-1}  floating_point?(double)
{-2}  typ′ = double
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[(range_double)]((range_floating_point(double)))(dt_double`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range_floating_point,

which is trivially true.
This completes the proof of `dt_floating_point_TCC2.1`.
`dt_floating_point_TCC2.2`:

```
{-1}  floating_point?(double)
{-2}  typ′ = double
------------------------------------------------------------------------
{1}   pod_data_type?[((range_floating_point(typ′)))](dt_double)
```

Adding type constraints for dt_double,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_floating_point_TCC2.2.1`:

```
{-1}  uninterpreted_data_type?(dt_double`uidt)
{-2}  ∀ (d: ((range_double)), a: Address):
        valid?(uidt(dt_double))(to_byte(dt_double)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_double))(l, a) ≡ up?(from_byte(dt_double)(l, a))
{-4}  ∀ (d: ((range_double)), a: Address):
        down(from_byte(dt_double)(to_byte(dt_double)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_double))(l, a1) ≡ valid?(uidt(dt_double))(l, a2)
{-6}  size(uidt(dt_double)) > 0
{-7}  floating_point?(double)
{-8}  typ′ = double
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(typ′))), a: Address):
        valid?(uidt(dt_double))(to_byte(dt_double)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_floating_point,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_floating_point_TCC2.2.1`.
`dt_floating_point_TCC2.2.2`:

```
{-1}  uninterpreted_data_type?(dt_double`uidt)
{-2}  ∀ (d: ((range_double)), a: Address):
        valid?(uidt(dt_double))(to_byte(dt_double)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_double))(l, a) ≡ up?(from_byte(dt_double)(l, a))
{-4}  ∀ (d: ((range_double)), a: Address):
        down(from_byte(dt_double)(to_byte(dt_double)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_double))(l, a1) ≡ valid?(uidt(dt_double))(l, a2)
{-6}  size(uidt(dt_double)) > 0
{-7}  floating_point?(double)
{-8}  typ′ = double
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(typ′))), a: Address):
        down(from_byte(dt_double)(to_byte(dt_double)(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_floating_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_floating_point_TCC2.2.2`.
Q.E.D.

C.43.22 Cpp_Integral_Types.dt_floating_point_TCC3

Terse proof for `dt_floating_point_TCC3`.
`dt_floating_point_TCC3:`

```
{1}  ∀ (typ: (floating_point?)):
       typ = longdouble ⊃
        (∀ (x: extended_real): (range_longdouble)(x) ≡ range_floating_point(typ)(x)) ∧
         (∀ (x: extended_real): (range_longdouble)(x) ≡ range_floating_point(typ)(x))
          ∧
          (∀ (x₁: [list[Byte], Address]):
               every[(range_longdouble)]
                    ((range_floating_point(typ)))(dt_longdouble`from_byte(x₁)))
           ∧ pod_data_type?[(range_floating_point(typ))](dt_longdouble)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of range_floating_point,
Applying propositional simplification,
we get 2 subgoals:
`dt_floating_point_TCC3.1:`

```
{-1}  floating_point?(longdouble)
{-2}  typ′ = longdouble
-------------------------------------------------------------------------
{1}  ∀ (x₁: [list[Byte], Address]):
       every[(range_longdouble)]
            ((range_floating_point(longdouble)))(dt_longdouble`from_byte(x₁))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range_floating_point,
which is trivially true.
This completes the proof of `dt_floating_point_TCC3.1`.
`dt_floating_point_TCC3.2:`

```
{-1}  floating_point?(longdouble)
{-2}  typ′ = longdouble
-------------------------------------------------------------------------
{1}  pod_data_type?[((range_floating_point(typ′)))](dt_longdouble)
```

Adding type constraints for dt_longdouble,
Expanding the definition of pod_data_type?,
Applying propositional simplification,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_floating_point_TCC3.2.1`:

```
{-1}  uninterpreted_data_type?(dt_longdouble`uidt)
{-2}  ∀ (d: ((range_longdouble)), a: Address):
        valid?(uidt(dt_longdouble))(to_byte(dt_longdouble)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_longdouble))(l, a) ≡ up?(from_byte(dt_longdouble)(l, a))
{-4}  ∀ (d: ((range_longdouble)), a: Address):
        down(from_byte(dt_longdouble)(to_byte(dt_longdouble)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_longdouble))(l, a1) ≡ valid?(uidt(dt_longdouble))(l, a2)
{-6}  size(uidt(dt_longdouble)) > 0
{-7}  floating_point?(longdouble)
{-8}  typ′ = longdouble
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(typ′))), a: Address):
        valid?(uidt(dt_longdouble))(to_byte(dt_longdouble)(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_floating_point,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_floating_point_TCC3.2.1`.
`dt_floating_point_TCC3.2.2`:

```
{-1}  uninterpreted_data_type?(dt_longdouble`uidt)
{-2}  ∀ (d: ((range_longdouble)), a: Address):
        valid?(uidt(dt_longdouble))(to_byte(dt_longdouble)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_longdouble))(l, a) ≡ up?(from_byte(dt_longdouble)(l, a))
{-4}  ∀ (d: ((range_longdouble)), a: Address):
        down(from_byte(dt_longdouble)(to_byte(dt_longdouble)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_longdouble))(l, a1) ≡ valid?(uidt(dt_longdouble))(l, a2)
{-6}  size(uidt(dt_longdouble)) > 0
{-7}  floating_point?(longdouble)
{-8}  typ′ = longdouble
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(typ′))), a: Address):
        down(from_byte(dt_longdouble)(to_byte(dt_longdouble)(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range_floating_point,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_floating_point_TCC3.2.2`.
Q.E.D.

C.43.23 Cpp_Integral_Types.range_integral_unsigned_TCC1

Terse proof for `range_integral_unsigned_TCC1`.
`range_integral_unsigned_TCC1`:

```
------------------------------------------------------------------------
{1}   ∀ (typ: (unsigned_integer?)): non_bool_integral?(typ)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_integral_unsigned_TCC1`.

Q.E.D.

C.43.24 Cpp_Integral_Types.range_integral_unsigned

Terse proof for `range_integral_unsigned`.
`range_integral_unsigned:`

{1}	∀ (typ: (unsigned_integer?)): (∀ (x: int): range_integral(typ)(x) ⊃ $x \geq 0$) ∧ is_finite[nat](range_integral(typ)) ∧ ¬ empty?[nat](range_integral(typ))

Repeatedly Skolemizing and flattening,
Case splitting on FORALL (x: int): range_integral(typ!1)(x) IMPLIES x >= 0,
we get 3 subgoals:
`range_integral_unsigned.1:`

{-1}	∀ (x: int): range_integral(typ′)(x) ⊃ $x \geq 0$
{-2}	unsigned_integer?(typ′)
{1}	(∀ (x: int): range_integral(typ′)(x) ⊃ $x \geq 0$) ∧ is_finite[nat](range_integral(typ′)) ∧ ¬ empty?[nat](range_integral(typ′))

Adding type constraints for range_integral(typ!1),
Applying propositional simplification,
we get 2 subgoals:
`range_integral_unsigned.1.1:`

{-1}	is_finite[int](range_integral(typ′))
{-2}	∀ (x: int): range_integral(typ′)(x) ⊃ $x \geq 0$
{-3}	unsigned_integer?(typ′)
{1}	is_finite[nat](range_integral(typ′))
{2}	empty?[int](range_integral(typ′))

Expanding the definition of is_finite,
which is trivially true.
This completes the proof of `range_integral_unsigned.1.1`.
`range_integral_unsigned.1.2:`

{-1}	empty?[nat](range_integral(typ′))
{-2}	is_finite[int](range_integral(typ′))
{-3}	∀ (x: int): range_integral(typ′)(x) ⊃ $x \geq 0$
{-4}	unsigned_integer?(typ′)
{1}	empty?[int](range_integral(typ′))

Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:
`range_integral_unsigned.1.2.1:`

{-1}	integer_pred(x')
{-2}	is_finite[int](range_integral(typ′))
{-3}	(x' ∈ range_integral(typ′))
{-4}	∀ (x: int): range_integral(typ′)(x) ⊃ $x \geq 0$
{-5}	unsigned_integer?(typ′)
{1}	(x' ∈ range_integral(typ′))

Expanding the definition of member,
which is trivially true.
This completes the proof of `range_integral_unsigned.1.2.1`.
`range_integral_unsigned.1.2.2`:

{-1}	integer_pred(x')
{-2}	is_finite[int](range_integral(typ$'$))
{-3}	(x' ∈ range_integral(typ$'$))
{-4}	∀ (x: int): range_integral(typ$'$)(x) ⊃ $x \geq 0$
{-5}	unsigned_integer?(typ$'$)
{1}	$x' \geq 0$

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of member,
which is trivially true.
This completes the proof of `range_integral_unsigned.1.2.2`.
`range_integral_unsigned.2`:

{-1}	unsigned_integer?(typ$'$)
{1}	∀ (x: int): range_integral(typ$'$)(x) ⊃ $x \geq 0$
{2}	(∀ (x: int): range_integral(typ$'$)(x) ⊃ $x \geq 0$) ∧ is_finite[nat](range_integral(typ$'$)) ∧ ¬ empty?[nat](range_integral(typ$'$))

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of unsigned_integer?,
Expanding the definition of range_integral,
Adding type constraints for range_uchar,
Adding type constraints for range_ulonglong,
Adding type constraints for range_ulong,
Adding type constraints for range_ushort,
Adding type constraints for range_uint,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_integral_unsigned.2`.
`range_integral_unsigned.3`:

{-1}	unsigned_integer?(typ$'$)
{1}	non_bool_integral?(typ$'$)
{2}	(∀ (x: int): range_integral(typ$'$)(x) ⊃ $x \geq 0$) ∧ is_finite[nat](range_integral(typ$'$)) ∧ ¬ empty?[nat](range_integral(typ$'$))

Expanding the definition of non_bool_integral?,
which is trivially true.
This completes the proof of `range_integral_unsigned.3`.
Q.E.D.

C.43.25 Cpp_Integral_Types.max_value_TCC1

Terse proof for `max_value_TCC1`.
`max_value_TCC1`:

{1}	∀ (typ: (non_bool_integral?)): ¬ empty?[real](extend[real, int, bool, FALSE](range_integral(typ)))

Repeatedly Skolemizing and flattening,
Adding type constraints for range_integral(typ!1),
Expanding the definition of empty?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `max_value_TCC1`.
Q.E.D.

C.43.26 Cpp_Integral_Types.max_value_TCC2

Terse proof for `max_value_TCC2`.

`max_value_TCC2:`

```
{1}  ∀ (typ: (non_bool_integral?)):
       rational_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ)))) ∧
         integer_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ))))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for max[real, <=](extend[real, int, bool, FALSE](range_integral(typ!1))),
we get 2 subgoals:

`max_value_TCC2.1:`

```
{-1}  extend[real, int, bool, FALSE]
        (range_integral(typ′))
        (max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))
{-2}  ∀ (x: real):
        extend[real, int, bool, FALSE](range_integral(typ′))(x) ⊃
          x ≤ max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))
{-3}  non_bool_integral?(typ′)
------------------------------------------------------------------------------
{1}   rational_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))) ∧
        integer_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))
```

Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `max_value_TCC2.1`.

`max_value_TCC2.2:`

```
{-1}  empty?[real](extend[real, int, bool, FALSE](range_integral(typ′)))
{-2}  non_bool_integral?(typ′)
------------------------------------------------------------------------------
{1}   rational_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))) ∧
        integer_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))
```

Using lemma max_value_TCC1,
This completes the proof of `max_value_TCC2.2`.
Q.E.D.

C.43.27 Cpp_Integral_Types.min_value_TCC1

Terse proof for `min_value_TCC1`.

`min_value_TCC1:`

{1} ∀ (typ: (non_bool_integral?)):
rational_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ)))) ∧
integer_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ))))

Repeatedly Skolemizing and flattening,
Adding type constraints for min[real, <=](extend[real, int, bool, FALSE](range_integral(typ!1))),
we get 2 subgoals:
`min_value_TCC1.1:`

{-1} extend[real, int, bool, FALSE]
(range_integral(typ′))
(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))
{-2} ∀ (x: real):
extend[real, int, bool, FALSE](range_integral(typ′))(x) ⊃
min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))) ≤ x
{-3} non_bool_integral?(typ′)

{1} rational_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))) ∧
integer_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))

Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `min_value_TCC1.1`.
`min_value_TCC1.2:`

{-1} empty?[real](extend[real, int, bool, FALSE](range_integral(typ′)))
{-2} non_bool_integral?(typ′)

{1} rational_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))) ∧
integer_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))

Using lemma max_value_TCC1,
This completes the proof of `min_value_TCC1.2`.
Q.E.D.

C.43.28 Cpp_Integral_Types.max_value_unsigned

Terse proof for `max_value_unsigned`.
`max_value_unsigned:`

{1} ∀ (typ: (unsigned_integer?)): max_value(typ) ≥ 0

Repeatedly Skolemizing and flattening,
Expanding the definition of max_value,
Using lemma range_integral_unsigned,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
we get 2 subgoals:

`max_value_unsigned.1:`

```
{-1}  range_integral(typ′)(max(extend[real, int, bool, FALSE](range_integral(typ′)))) ⊃
        max(extend[real, int, bool, FALSE](range_integral(typ′))) ≥ 0
{-2}  unsigned_integer?(typ′)
------------------------------------------------------------------------------
{1}   empty?[nat](range_integral(typ′))
{2}   max(extend[real, int, bool, FALSE](range_integral(typ′))) ≥ 0
```

Adding type constraints for max(extend[real, int, bool, FALSE](range_integral(typ!1))),
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `max_value_unsigned.1`.

`max_value_unsigned.2:`

```
{-1}  unsigned_integer?(typ′)
------------------------------------------------------------------------------
{1}   rational_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))) ∧
        integer_pred(max[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))
{2}   empty?[nat](range_integral(typ′))
{3}   max(extend[real, int, bool, FALSE](range_integral(typ′))) ≥ 0
```

Using lemma max_value_TCC2,
This completes the proof of `max_value_unsigned.2`.
Q.E.D.

C.43.29 Cpp_Integral_Types.min_value_unsigned

Terse proof for `min_value_unsigned`.

`min_value_unsigned:`

```
------------------------------------------------------------------------------
{1}   ∀ (typ: (unsigned_integer?)): min_value(typ) ≥ 0
```

Expanding the definition of min_value,
Repeatedly Skolemizing and flattening,
Using lemma range_integral_unsigned,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
we get 2 subgoals:

`min_value_unsigned.1:`

```
{-1}  range_integral(typ′)(min(extend[real, int, bool, FALSE](range_integral(typ′)))) ⊃
        min(extend[real, int, bool, FALSE](range_integral(typ′))) ≥ 0
{-2}  unsigned_integer?(typ′)
------------------------------------------------------------------------------
{1}   empty?[nat](range_integral(typ′))
{2}   min(extend[real, int, bool, FALSE](range_integral(typ′))) ≥ 0
```

Adding type constraints for min(extend[real, int, bool, FALSE](range_integral(typ!1))),
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `min_value_unsigned.1`.

`min_value_unsigned.2:`

{-1} unsigned_integer?(typ′)

{1} rational_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′)))) ∧
integer_pred(min[real, ≤](extend[real, int, bool, FALSE](range_integral(typ′))))
{2} empty?[nat](range_integral(typ′))
{3} min(extend[real, int, bool, FALSE](range_integral(typ′))) ≥ 0

Using lemma min_value_TCC1,
This completes the proof of `min_value_unsigned.2`.
Q.E.D.

C.43.30 Cpp_Integral_Types.max_value_bits_TCC1

Terse proof for `max_value_bits_TCC1`.
`max_value_bits_TCC1:`

{1} well_founded?(λ (x: (cv(unsigned_integer?)), y: (cv(unsigned_integer?))):
restrict
[[Cpp_Type_, Cpp_Type_],
[(cv(unsigned_integer?)), (cv(unsigned_integer?))], boolean]
(≪)(x, y))

Using lemma well_founded_restrict[Cpp_Type_, (cv(unsigned_integer?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `max_value_bits_TCC1`.
Q.E.D.

C.43.31 Cpp_Integral_Types.max_value_bits_TCC2

Terse proof for `max_value_bits_TCC2`.
`max_value_bits_TCC2:`

{1} ∀ (typ: (cv(unsigned_integer?)), t: Cpp_Type_):
typ = const(t) ⊃ cv(unsigned_integer?)(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `max_value_bits_TCC2`.
Q.E.D.

C.43.32 Cpp_Integral_Types.max_value_bits_TCC3

Terse proof for `max_value_bits_TCC3`.
`max_value_bits_TCC3:`

{1} ∀ (typ: (cv(unsigned_integer?)), t: Cpp_Type_):
typ = const(t) ⊃
restrict
[[Cpp_Type_, Cpp_Type_], [(cv(unsigned_integer?)), (cv(unsigned_integer?))],
boolean]
(≪)(t, typ)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `max_value_bits_TCC3`.

Q.E.D.

C.43.33 Cpp_Integral_Types.max_value_bits_TCC4

Terse proof for `max_value_bits_TCC4`.

`max_value_bits_TCC4:`

```
{1}   ∀ (typ: (cv(unsigned_integer?)), t: Cpp_Type_):
        typ = volatile(t) ⊃ cv(unsigned_integer?)(t)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `max_value_bits_TCC4`.

Q.E.D.

C.43.34 Cpp_Integral_Types.max_value_bits_TCC5

Terse proof for `max_value_bits_TCC5`.

`max_value_bits_TCC5:`

```
{1}   ∀ (typ: (cv(unsigned_integer?)), t: Cpp_Type_):
        typ = volatile(t) ⊃
         restrict
              [[Cpp_Type_, Cpp_Type_], [(cv(unsigned_integer?)), (cv(unsigned_integer?))],
                boolean]
              (≪)(t, typ)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `max_value_bits_TCC5`.

Q.E.D.

C.43.35 Cpp_Integral_Types.max_value_bits_TCC6

Terse proof for `max_value_bits_TCC6`.

`max_value_bits_TCC6:`

{1} ∀ (typ: (cv(unsigned_integer?))):
¬ (schar?(typ) ∨
char?(typ) ∨
short?(typ) ∨
int?(typ) ∨
long?(typ) ∨
longlong?(typ) ∨
wchar_t?(typ) ∨
bool?(typ) ∨
float?(typ) ∨
double?(typ) ∨
longdouble?(typ) ∨
void?(typ) ∨
array?(typ) ∨
function?(typ) ∨
pointer?(typ) ∨
reference?(typ) ∨
class?(typ) ∨
union?(typ) ∨
bitfield?(typ) ∨ enum?(typ) ∨ pointer_to_member?(typ))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `max_value_bits_TCC6`.
Q.E.D.

C.44 Proofs for Cpp_Types (types.pvs)

C.44.1 Cpp_Types.alignment_TCC1

Terse proof for `alignment_TCC1`.
`alignment_TCC1:`

{1} well_founded?(λ (x: Cpp_Type, y: Cpp_Type):
restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean]
(≪)(x, y))

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Type],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `alignment_TCC1`.
Q.E.D.

C.44.2 Cpp_Types.alignment_TCC2

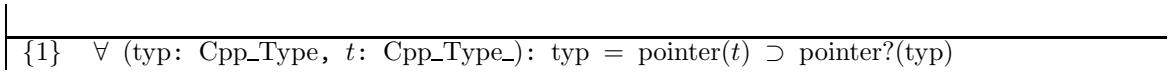
Terse proof for `alignment_TCC2`.
`alignment_TCC2:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_, s: nat): typ = array_type(t, s) ⊃ array?(typ)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC2`.
Q.E.D.

C.44.3 Cpp_Types.alignment_TCC3

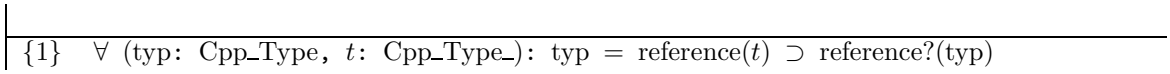
Terse proof for `alignment_TCC3`.
`alignment_TCC3`:



Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC3`.
Q.E.D.

C.44.4 Cpp_Types.alignment_TCC4

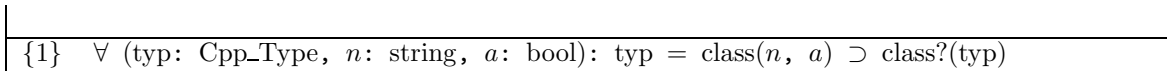
Terse proof for `alignment_TCC4`.
`alignment_TCC4`:



Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC4`.
Q.E.D.

C.44.5 Cpp_Types.alignment_TCC5

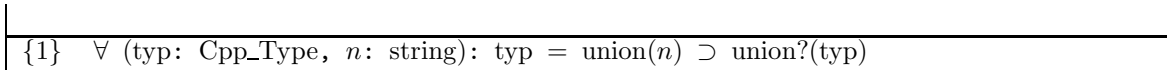
Terse proof for `alignment_TCC5`.
`alignment_TCC5`:



Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC5`.
Q.E.D.

C.44.6 Cpp_Types.alignment_TCC6

Terse proof for `alignment_TCC6`.
`alignment_TCC6`:



Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC6`.
Q.E.D.

C.44.7 Cpp_Types.alignment_TCC7

Terse proof for `alignment_TCC7`.
`alignment_TCC7`:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_, bo: nat, s: posnat):
typ = bitfield(t, bo, s) ⊃ bitfield?(typ)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC7`.
Q.E.D.

C.44.8 Cpp_Types.alignment_TCC8

Terse proof for `alignment_TCC8`.
`alignment_TCC8`:

{1} ∀ (typ: Cpp_Type, n: string, t: Cpp_Type_, m: posnat, c: [below[m] → int]):
typ = enum(n, t, m, c) ⊃ enum?(typ)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC8`.
Q.E.D.

C.44.9 Cpp_Types.alignment_TCC9

Terse proof for `alignment_TCC9`.
`alignment_TCC9`:

{1} ∀ (typ: Cpp_Type, t: (class?), m: Cpp_Type_):
typ = pointer_to_member(t, m) ⊃ pointer_to_member?(typ)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC9`.
Q.E.D.

C.44.10 Cpp_Types.alignment_TCC10

Terse proof for `alignment_TCC10`.
`alignment_TCC10`:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_): typ = const(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC10`.
Q.E.D.

C.44.11 Cpp_Types.alignment_TCC11

Terse proof for `alignment_TCC11`.

`alignment_TCC11`:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
       typ = const(t) ⊃
         restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean](≪)(t, typ)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of restrict,
Replacing using formula -2,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `alignment_TCC11`.
Q.E.D.

C.44.12 Cpp_Types.alignment_TCC12

Terse proof for `alignment_TCC12`.

`alignment_TCC12`:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_): typ = volatile(t) ⊃ Cpp_Type?(t)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: (t!2),
Expanding the definition of subterm,
Replacing using formula -2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `alignment_TCC12`.
Q.E.D.

C.44.13 Cpp_Types.alignment_TCC13

Terse proof for `alignment_TCC13`.

`alignment_TCC13`:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = volatile(t) ⊃
restrict[[Cpp_Type_, Cpp_Type_], [Cpp_Type, Cpp_Type], boolean](≪)(t, typ)

Repeatedly Skolemizing and flattening,
Expanding the definition of restrict,
Replacing using formula -2,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `alignment_TCC13`.
Q.E.D.

C.44.14 Cpp_Types.pointer_to_aligned_object_TCC1

Terse proof for `pointer_to_aligned_object_TCC1`.

`pointer_to_aligned_object_TCC1`:

{1} ∀ (p: Cpp_Subtype(cv(pointer?)), ptr_val: (range_pointer(cv_base(p)))):
not_null?(p)(ptr_val) ⊃
array?(cv_base(p)) ∨
pointer?(cv_base(p)) ∨
reference?(cv_base(p)) ∨
bitfield?(cv_base(p)) ∨
enum?(cv_base(p)) ∨
pointer_to_member?(cv_base(p)) ∨ const?(cv_base(p)) ∨ volatile?(cv_base(p))

Repeatedly Skolemizing and flattening,
Keeping (-2 2) and hiding *,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pointer_to_aligned_object_TCC1`.
Q.E.D.

C.44.15 Cpp_Types.pointer_to_aligned_object_TCC2

Terse proof for `pointer_to_aligned_object_TCC2`.

`pointer_to_aligned_object_TCC2`:

{1} ∀ (p: Cpp_Subtype(cv(pointer?)), ptr_val: (range_pointer(cv_base(p)))):
not_null?(p)(ptr_val) ⊃ Cpp_Type?(typ(cv_base(p)))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: (t!1),
Splitting conjunctions,
we get 2 subgoals:

`pointer_to_aligned_object_TCC2.1:`

{-1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

{-2} $\text{cv(pointer?)}(p')$

{-3} $\text{range\_pointer}(\text{cv\_base}(p'))(\text{ptr\_val}')$

{-4} $\text{not\_null?}(p')(\text{ptr\_val}')$

{-5} $\text{subterm}(t', \text{typ}(\text{cv\_base}(p')))$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

which is trivially true.
This completes the proof of `pointer_to_aligned_object_TCC2.1`.
`pointer_to_aligned_object_TCC2.2`:

{-1} cv(pointer?)(p')
{-2} range_pointer(cv_base(p'))(ptr_val$'$)
{-3} not_null?(p')(ptr_val$'$)
{-4} subterm(t', typ(cv_base(p')))

{1} subterm(t', p')
{2} no_pointers_to_bitfield?(t') ∧
no_pointers_to_references?(t') ∧
no_cv_references?(t') ∧
no_reference_to_reference?(t') ∧
no_reference_to_bitfields?(t') ∧
no_pointer_to_member_to_reference?(t') ∧
no_pointer_to_member_to_cv_void?(t') ∧
no_cv_void_parameter?(t') ∧
no_array_of_references?(t') ∧
no_array_of_cv_void?(t') ∧
no_array_of_function?(t') ∧
no_array_of_abstract_class?(t') ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧
no_array_return_type?(t') ∧
no_function_return_type?(t') ∧
bitfield_underlying_integral_or_enum_type?(t') ∧
cv_array?(t') ∧
enum_underlying_integral?(t') ∧
enum_constants?(t') ∧
const_volatile?(t') ∧
const_stutter?(t') ∧
volatile_stutter?(t') ∧
no_cv_class?(t') ∧
no_cv_union?(t') ∧
no_cv_function?(t') ∧
no_reference_to_void?(t') ∧
no_cv_void?(t') ∧ no_array_of_bitfields?(t')

Hiding formulas: 2,
Using lemma subterm_cv_base,
Using lemma subterm_transitive,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma cv_base_result,
we get 2 subgoals:
`pointer_to_aligned_object_TCC2.2.1`:

{-1} pointer?(cv_base(p'))
{-2} subterm(cv_base(p'), p')
{-3} cv(pointer?)(p')
{-4} range_pointer(cv_base(p'))(ptr_val$'$)
{-5} not_null?(p')(ptr_val$'$)
{-6} subterm(t', typ(cv_base(p')))

{1} subterm(t', cv_base(p'))
{2} subterm(t', p')

Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pointer_to_aligned_object_TCC2.2.1`.
`pointer_to_aligned_object_TCC2.2.2`:

```
{-1}  subterm(cv_base(p'), p')
{-2}  cv(pointer?)(p')
{-3}  range_pointer(cv_base(p'))(ptr_val')
{-4}  not_null?(p')(ptr_val')
{-5}  subterm(t', typ(cv_base(p')))
------------------------------------------------------------
{1}   (pointer? ⊆ interpreted?)
{2}   subterm(t', cv_base(p'))
{3}   subterm(t', p')
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pointer_to_aligned_object_TCC2.2.2`.
Q.E.D.

C.44.16 Cpp_Types.pointer_to_aligned_object_TCC3

Terse proof for `pointer_to_aligned_object_TCC3`.
`pointer_to_aligned_object_TCC3`:

```
------------------------------------------------------------
{1}   ∀ (p: Cpp_Subtype(cv(pointer?)), ptr_val: (range_pointer(cv_base(p)))):
        not_null?(p)(ptr_val) ⊃ up?[Memory_Address](address_of(p)(ptr_val))
```

Repeatedly Skolemizing and flattening,
Rewriting using address_of_spec, matching in *,
This completes the proof of `pointer_to_aligned_object_TCC3`.
Q.E.D.

C.44.17 Cpp_Types.align_array_of_type_TCC1

Terse proof for `align_array_of_type_TCC1`.
`align_array_of_type_TCC1`:

```
------------------------------------------------------------
{1}   ∀ (a: Cpp_Subtype(array?)):
        array?(a) ∨
          pointer?(a) ∨
            reference?(a) ∨
              bitfield?(a) ∨ enum?(a) ∨ pointer_to_member?(a) ∨ const?(a) ∨ volatile?(a)
```

Repeatedly Skolemizing and flattening,
This completes the proof of `align_array_of_type_TCC1`.
Q.E.D.

C.44.18 Cpp_Types.align_array_of_type_TCC2

Terse proof for `align_array_of_type_TCC2`.

`align_array_of_type_TCC2`:

{1} ∀ (a: Cpp_Subtype(array?)): Cpp_Type?(typ(a))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: (t!1),
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `align_array_of_type_TCC2`.
Q.E.D.

C.44.19 Cpp_Types.range_TCC1

Terse proof for `range_TCC1`.

`range_TCC1`:

{1} well_founded?(λ (x: Cpp_Subtype(cv(non_bool_integral_enum?)),
y: Cpp_Subtype(cv(non_bool_integral_enum?))):
restrict
[[Cpp_Type_, Cpp_Type_],
[Cpp_Subtype(cv(non_bool_integral_enum?)),
Cpp_Subtype(cv(non_bool_integral_enum?))],
boolean]
(≪)(x, y))

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(non_bool_integral_enum?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `range_TCC1`.
Q.E.D.

C.44.20 Cpp_Types.range_TCC2

Terse proof for `range_TCC2`.

`range_TCC2`:

{1} ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), n: string, t: Cpp_Type_, m: posnat,
c: [below[m] → int]):
typ = enum(n, t, m, c) ⊃ enum?(typ)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_TCC2`.
Q.E.D.

C.44.21 Cpp_Types.range_TCC3

Terse proof for `range_TCC3`.

`range_TCC3:`

```
|-------
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), t: Cpp_Type_):
       typ = const(t) ⊃ Cpp_Type?(t) ∧ cv(non_bool_integral_enum?)(t)
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
`range_TCC3.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(non_bool_integral_enum?)(typ′)
{-3}  typ′ = const(t′)
|-------
{1}   Cpp_Type?(t′)
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Replacing using formula -4,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_TCC3.1`.
`range_TCC3.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(non_bool_integral_enum?)(typ′)
{-3}  typ′ = const(t′)
|-------
{1}   cv(non_bool_integral_enum?)(t′)
```

Replacing using formula -3,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_TCC3.2`.
Q.E.D.

C.44.22 Cpp_Types.range_TCC4

Terse proof for `range_TCC4`.
`range_TCC4:`

```
|-------
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), t: Cpp_Type_):
       typ = const(t) ⊃
        restrict
             [[Cpp_Type_, Cpp_Type_],
               [Cpp_Subtype(cv(non_bool_integral_enum?)),
                 Cpp_Subtype(cv(non_bool_integral_enum?))],
               boolean]
             (≪)(t, typ)
```

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `range_TCC4`.

Q.E.D.

C.44.23 Cpp_Types.range_TCC5

Terse proof for `range_TCC5`.

`range_TCC5:`

{1} ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), t: Cpp_Type_):
typ = volatile(t) ⊃ Cpp_Type?(t) ∧ cv(non_bool_integral_enum?)(t)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`range_TCC5.1:`

{-1} Cpp_Type?(typ′)
{-2} cv(non_bool_integral_enum?)(typ′)
{-3} typ′ = volatile(t')

{1} Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Replacing using formula -4,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_TCC5.1`.

`range_TCC5.2:`

{-1} Cpp_Type?(typ′)
{-2} cv(non_bool_integral_enum?)(typ′)
{-3} typ′ = volatile(t')

{1} cv(non_bool_integral_enum?)(t')

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_TCC5.2`.
Q.E.D.

C.44.24 Cpp_Types.range_TCC6

Terse proof for `range_TCC6`.

`range_TCC6:`

{1} ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), t: Cpp_Type_):
typ = volatile(t) ⊃
restrict
[[Cpp_Type_, Cpp_Type_],
[Cpp_Subtype(cv(non_bool_integral_enum?)),
Cpp_Subtype(cv(non_bool_integral_enum?))],
boolean]
(≪)(t, typ)

Repeatedly Skolemizing and flattening,

Replacing using formula -3,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_TCC6`.
Q.E.D.

C.44.25 Cpp_Types.range_TCC7

Terse proof for `range_TCC7`.
`range_TCC7:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
       ¬ volatile?(typ) ∧ ¬ const?(typ) ∧ ¬ enum?(typ) ⊃ non_bool_integral?(typ)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of v,
Expanding the definition of c,
Expanding the definition of non_bool_integral_enum?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `range_TCC7`.
Q.E.D.

C.44.26 Cpp_Types.range_cv_base_TCC1

Terse proof for `range_cv_base_TCC1`.
`range_cv_base_TCC1:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
       cv(non_bool_integral_enum?)(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of cv,
Expanding the definition of c,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of v,
Hiding formulas: (2 3 4),
Applying propositional simplification,
we get 5 subgoals:
`range_cv_base_TCC1.1:`

```
{-1}  non_bool_integral_enum?(typ′)
{1}   non_bool_integral_enum?(cv_base(typ′))
```

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`range_cv_base_TCC1.1.1:`

```
{-1}  non_bool_integral_enum?(typ′)
{-2}  const?(typ′)
{1}   non_bool_integral_enum?(cv_base(typ(typ′)))
```

Hiding formulas: 1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.1.1`.
`range_cv_base_TCC1.1.2:`

```
{-1}  non_bool_integral_enum?(typ')
{-2}  volatile?(typ')
------------------------------------------------------------
{1}   non_bool_integral_enum?(cv_base(typ(typ')))
```

Hiding formulas: 1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.1.2`.
`range_cv_base_TCC1.2:`

```
{-1}  const?(typ')
{-2}  non_bool_integral_enum?(typ(typ'))
------------------------------------------------------------
{1}   non_bool_integral_enum?(cv_base(typ'))
```

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`range_cv_base_TCC1.2.1:`

```
{-1}  const?(typ')
{-2}  non_bool_integral_enum?(typ(typ'))
{-3}  const?(typ(typ'))
------------------------------------------------------------
{1}   non_bool_integral_enum?(cv_base(typ(typ(typ'))))
```

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.2.1`.
`range_cv_base_TCC1.2.2:`

```
{-1}  const?(typ')
{-2}  non_bool_integral_enum?(typ(typ'))
{-3}  volatile?(typ(typ'))
------------------------------------------------------------
{1}   non_bool_integral_enum?(cv_base(typ(typ(typ'))))
```

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.2.2`.
`range_cv_base_TCC1.3:`

```
{-1}  non_bool_integral_enum?(typ')
------------------------------------------------------------
{1}   non_bool_integral_enum?(cv_base(typ'))
```

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`range_cv_base_TCC1.3.1:`

```
{-1}  non_bool_integral_enum?(typ')
{-2}  const?(typ')
------------------------------------------------------------
{1}   non_bool_integral_enum?(cv_base(typ(typ')))
```

Keeping (-2 -1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.3.1`.
`range_cv_base_TCC1.3.2`:

{-1}	non_bool_integral_enum?(typ′)
{-2}	volatile?(typ′)
{1}	non_bool_integral_enum?(cv_base(typ(typ′)))

Keeping (-2 -1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.3.2`.
`range_cv_base_TCC1.4`:

{-1}	volatile?(typ′)
{-2}	non_bool_integral_enum?(typ(typ′))
{1}	non_bool_integral_enum?(cv_base(typ′))

Expanding the definition of cv_base,
Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`range_cv_base_TCC1.4.1`:

{-1}	volatile?(typ′)
{-2}	non_bool_integral_enum?(typ(typ′))
{-3}	const?(typ(typ′))
{1}	non_bool_integral_enum?(cv_base(typ(typ(typ′))))

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.4.1`.
`range_cv_base_TCC1.4.2`:

{-1}	volatile?(typ′)
{-2}	non_bool_integral_enum?(typ(typ′))
{-3}	volatile?(typ(typ′))
{1}	non_bool_integral_enum?(cv_base(typ(typ(typ′))))

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.4.2`.
`range_cv_base_TCC1.5`:

{-1}	const?(typ′)
{-2}	volatile?(typ(typ′))
{-3}	non_bool_integral_enum?(typ(typ(typ′)))
{1}	non_bool_integral_enum?(cv_base(typ′))

Expanding the definition of cv_base,
Expanding the definition of cv_base,
Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`range_cv_base_TCC1.5.1:`

{-1}	const?(typ′)
{-2}	volatile?(typ(typ′))
{-3}	non_bool_integral_enum?(typ(typ(typ′)))
{-4}	const?(typ(typ(typ′)))
{1}	non_bool_integral_enum?(cv_base(typ(typ′)))

Keeping (-4 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.5.1`.
`range_cv_base_TCC1.5.2:`

{-1}	const?(typ′)
{-2}	volatile?(typ(typ′))
{-3}	non_bool_integral_enum?(typ(typ(typ′)))
{-4}	volatile?(typ(typ(typ′)))
{1}	non_bool_integral_enum?(cv_base(typ(typ(typ(typ′)))))

Keeping (-4 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base_TCC1.5.2`.
Q.E.D.

C.44.27 Cpp_Types.range_cv_base

Terse proof for `range_cv_base`.
`range_cv_base:`

{1}	∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))): range(typ) = range(cv_base(typ))

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`range_cv_base.1:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral_enum?(typ′)
{1}	range(typ′) = range(cv_base(typ′))

Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`range_cv_base.1.1:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral_enum?(typ′)
{-3}	const?(typ′)
{1}	range(typ′) = range(cv_base(typ(typ′)))

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.1.1`.
`range_cv_base.1.2:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral_enum?(typ′)
{-3}	volatile?(typ′)
{1}	range(typ′) = range(cv_base(typ(typ′)))

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base.1.2`.
`range_cv_base.2:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{1}	range(typ′) = range(cv_base(typ′))

Expanding the definition of cv_base,
Expanding the definition of range,
Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`range_cv_base.2.1:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{-4}	const?(typ(typ′))
{1}	range(typ(typ′)) = range(cv_base(typ(typ(typ′))))

Keeping (-3 -4) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base.2.1`.
`range_cv_base.2.2:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{-4}	volatile?(typ(typ′))
{1}	range(typ(typ′)) = range(cv_base(typ(typ(typ′))))

Keeping (-3 -4) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base.2.2`.
`range_cv_base.3:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	volatile?(typ(typ′))
{-4}	non_bool_integral_enum?(typ(typ(typ′)))
{1}	range(typ′) = range(cv_base(typ′))

Expanding the definition of cv_base,
Expanding the definition of cv_base,
Expanding the definition of range,
Expanding the definition of range,

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`range_cv_base.3.1:`

```
{-1}  Cpp_Type?(typ')
{-2}  const?(typ')
{-3}  volatile?(typ(typ'))
{-4}  non_bool_integral_enum?(typ(typ(typ')))
{-5}  const?(typ(typ(typ')))
------------------------------------------------------------------------
{1}   range(typ(typ(typ'))) = range(cv_base(typ(typ(typ(typ')))))
```

Keeping (-4 -5) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.3.1`.

`range_cv_base.3.2:`

```
{-1}  Cpp_Type?(typ')
{-2}  const?(typ')
{-3}  volatile?(typ(typ'))
{-4}  non_bool_integral_enum?(typ(typ(typ')))
{-5}  volatile?(typ(typ(typ')))
------------------------------------------------------------------------
{1}   range(typ(typ(typ'))) = range(cv_base(typ(typ(typ(typ')))))
```

Keeping (-4 -5) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.3.2`.

`range_cv_base.4:`

```
{-1}  Cpp_Type?(typ')
{-2}  non_bool_integral_enum?(typ(typ'))
{-3}  volatile?(typ')
------------------------------------------------------------------------
{1}   range(typ') = range(cv_base(typ'))
```

Expanding the definition of cv_base,

Expanding the definition of cv_base,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`range_cv_base.4.1:`

```
{-1}  Cpp_Type?(typ')
{-2}  non_bool_integral_enum?(typ(typ'))
{-3}  volatile?(typ')
{-4}  const?(typ(typ'))
------------------------------------------------------------------------
{1}   range(typ(typ')) = range(cv_base(typ(typ(typ'))))
```

Keeping (-2 -4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_cv_base.4.1`.

`range_cv_base.4.2:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral_enum?(typ(typ′))
{-3}	volatile?(typ′)
{-4}	volatile?(typ(typ′))
{1}	range(typ(typ′)) = range(cv_base(typ(typ(typ′))))

Keeping (-2 -4) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_cv_base.4.2`.
Q.E.D.

C.44.28 Cpp_Types.dt_non_bool_integral_enum_TCC1

Terse proof for `dt_non_bool_integral_enum_TCC1`.

`dt_non_bool_integral_enum_TCC1:`

{1}	∀ (typ: Cpp_Subtype(non_bool_integral_enum?), n: string, t: Cpp_Type_, m: posnat, c: [below[m] → int]): typ = enum(n, t, m, c) ⊃ enum?(typ)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC1`.
Q.E.D.

C.44.29 Cpp_Types.dt_non_bool_integral_enum_TCC2

Terse proof for `dt_non_bool_integral_enum_TCC2`.

`dt_non_bool_integral_enum_TCC2:`

{1}	∀ (typ: Cpp_Subtype(non_bool_integral_enum?), n: string, t: Cpp_Type_, m: posnat, c: [below[m] → int]): typ = enum(n, t, m, c) ⊃ (∀ (x: int): range_enum(typ)(x) ≡ range(typ)(x)) ∧ (∀ (x: int): range_enum(typ)(x) ≡ range(typ)(x)) ∧ (∀ (x_1: [list[Byte], Address]): every[((range_enum(typ)))]((range(typ)))(dt_enum(typ)‘from_byte(x_1))) ∧ interpreted_data_type?[(range(typ))](dt_enum(typ))

Repeatedly Skolemizing and flattening,
Replacing using formula -4,
Expanding the definition of range,
Adding type constraints for dt_enum(enum(n!1, t!1, m!1, c!1)),
we get 2 subgoals:

`dt_non_bool_integral_enum_TCC2.1:`

```
{-1}  pod_data_type?[((range_enum(enum(n', t', m', c'))))]
          (dt_enum(enum(n', t', m', c')))
{-2}  Cpp_Type?(enum(n', t', m', c'))
{-3}  non_bool_integral_enum?(enum(n', t', m', c'))
{-4}  m' > 0
{-5}  typ' = enum(n', t', m', c')
------------------------------------------------------------------------
{1}   (∀ (x1: [list[Byte], Address]):
          every[((range_enum(typ')))]
              (range_enum(enum(n', t', m', c')))
              (dt_enum(enum(n', t', m', c'))`from_byte(x1)))
       ∧ interpreted_data_type?[((range(typ')))](dt_enum(enum(n', t', m', c')))
```

Applying propositional simplification,

we get 2 subgoals:

`dt_non_bool_integral_enum_TCC2.1.1:`

```
{-1}  pod_data_type?[((range_enum(enum(n', t', m', c'))))]
          (dt_enum(enum(n', t', m', c')))
{-2}  Cpp_Type?(enum(n', t', m', c'))
{-3}  non_bool_integral_enum?(enum(n', t', m', c'))
{-4}  m' > 0
{-5}  typ' = enum(n', t', m', c')
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[((range_enum(typ')))]
            (range_enum(enum(n', t', m', c')))
            (dt_enum(enum(n', t', m', c'))`from_byte(x1))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_TCC2.1.1`.

`dt_non_bool_integral_enum_TCC2.1.2:`

```
{-1}  pod_data_type?[((range_enum(enum(n', t', m', c'))))]
          (dt_enum(enum(n', t', m', c')))
{-2}  Cpp_Type?(enum(n', t', m', c'))
{-3}  non_bool_integral_enum?(enum(n', t', m', c'))
{-4}  m' > 0
{-5}  typ' = enum(n', t', m', c')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range(typ')))](dt_enum(enum(n', t', m', c')))
```

Expanding the definition of pod_data_type?,

Applying disjunctive simplification to flatten sequent,

Keeping (-1 -7 1) and hiding *,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_non_bool_integral_enum_TCC2.1.2.1:`

```
{-1}  uninterpreted_data_type?(dt_enum(enum(n', t', m', c'))`uidt)
{-2}  ∀ (d: ((range_enum(enum(n', t', m', c')))), a: Address):
        valid?(uidt(dt_enum(enum(n', t', m', c'))))
            (to_byte(dt_enum(enum(n', t', m', c')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_enum(enum(n', t', m', c'))))(l, a) ≡
          up?(from_byte(dt_enum(enum(n', t', m', c')))(l, a))
{-4}  ∀ (d: ((range_enum(enum(n', t', m', c')))), a: Address):
        down(from_byte(dt_enum(enum(n', t', m', c'))))
                    (to_byte(dt_enum(enum(n', t', m', c')))(d, a), a))
          = d
{-5}  typ' = enum(n', t', m', c')
-------------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        valid?(uidt(dt_enum(enum(n', t', m', c'))))
            (to_byte(dt_enum(enum(n', t', m', c')))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_TCC2.1.2.1`.

`dt_non_bool_integral_enum_TCC2.1.2.2:`

```
{-1}  uninterpreted_data_type?(dt_enum(enum(n', t', m', c'))`uidt)
{-2}  ∀ (d: ((range_enum(enum(n', t', m', c')))), a: Address):
        valid?(uidt(dt_enum(enum(n', t', m', c'))))
            (to_byte(dt_enum(enum(n', t', m', c')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_enum(enum(n', t', m', c'))))(l, a) ≡
          up?(from_byte(dt_enum(enum(n', t', m', c')))(l, a))
{-4}  ∀ (d: ((range_enum(enum(n', t', m', c')))), a: Address):
        down(from_byte(dt_enum(enum(n', t', m', c'))))
                    (to_byte(dt_enum(enum(n', t', m', c')))(d, a), a))
          = d
{-5}  typ' = enum(n', t', m', c')
-------------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        down(from_byte(dt_enum(enum(n', t', m', c'))))
                    (to_byte(dt_enum(enum(n', t', m', c')))(d, a), a))
          = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_TCC2.1.2.2`.

`dt_non_bool_integral_enum_TCC2.2:`

```
{-1}  Cpp_Type?(enum(n', t', m', c'))
{-2}  non_bool_integral_enum?(enum(n', t', m', c'))
{-3}  m' > 0
{-4}  typ' = enum(n', t', m', c')
|-------
{1}   Cpp_Type?(enum(n', t', m', c'))
{2}   (∀ (x1: [list[Byte], Address]):
         every[((range_enum(typ')))]
              (range_enum(enum(n', t', m', c')))
              (dt_enum(enum(n', t', m', c'))`from_byte(x1)))
       ∧ interpreted_data_type?[((range(typ')))](dt_enum(enum(n', t', m', c')))
```

which is trivially true.
This completes the proof of `dt_non_bool_integral_enum_TCC2.2`.
Q.E.D.

C.44.30 Cpp_Types.dt_non_bool_integral_enum_TCC3

Terse proof for `dt_non_bool_integral_enum_TCC3`.
`dt_non_bool_integral_enum_TCC3:`

```
|-------
{1}   ∀ (typ: Cpp_Subtype(non_bool_integral_enum?)): ¬ enum?(typ) ⊃ non_bool_integral?(typ)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC3`.
Q.E.D.

C.44.31 Cpp_Types.dt_non_bool_integral_enum_TCC4

Terse proof for `dt_non_bool_integral_enum_TCC4`.
`dt_non_bool_integral_enum_TCC4:`

```
|-------
{1}   ∀ (typ: Cpp_Subtype(non_bool_integral_enum?)):
        ¬ enum?(typ) ⊃
          (∀ (x: int): range_integral(typ)(x) ≡ range(typ)(x)) ∧
            (∀ (x: int): range_integral(typ)(x) ≡ range(typ)(x)) ∧
              (∀ (x1: [list[Byte], Address]):
                   every[((range_integral(typ)))]
                        ((range(typ)))(dt_integral(typ)`from_byte(x1)))
                ∧ interpreted_data_type?[(range(typ))](dt_integral(typ))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of non_bool_integral_enum?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_non_bool_integral_enum_TCC4.1:`

```
{-1}  Cpp_Type?(typ')
{-2}  non_bool_integral?(typ')
|-------
{1}   enum?(typ')
{2}   interpreted_data_type?[((range(typ')))](dt_integral(typ'))
```

Adding type constraints for dt_integral(typ!1),
Using lemma pod_is_interpreted_data[(range_integral(typ′))],
Expanding the definition of interpreted_data_type?,
Keeping (-1 -4 2) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`dt_non_bool_integral_enum_TCC4.1.1:`

{-1} uninterpreted_data_type?(dt_integral(typ′)‘uidt)
{-2} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(typ′)))(l, a) ≡
up?(from_byte(dt_integral(typ′))(l, a))
{-4} ∀ (d: ((range_integral(typ′))), a: Address):
down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(d, a), a)) = d
{-5} non_bool_integral?(typ′)

{1} ∀ (d: ((range(typ′))), a: Address):
valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.1.1`.
`dt_non_bool_integral_enum_TCC4.1.2:`

{-1} uninterpreted_data_type?(dt_integral(typ′)‘uidt)
{-2} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(typ′)))(l, a) ≡
up?(from_byte(dt_integral(typ′))(l, a))
{-4} ∀ (d: ((range_integral(typ′))), a: Address):
down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(d, a), a)) = d
{-5} non_bool_integral?(typ′)

{1} ∀ (d: ((range(typ′))), a: Address):
down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Expanding the definition of range,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.1.2`.
`dt_non_bool_integral_enum_TCC4.2:`

{-1} Cpp_Type?(typ′)
{-2} non_bool_integral?(typ′)

{1} enum?(typ′)
{2} ∀ (x_1: [list[Byte], Address]):
every[((range_integral(typ′)))]
((range(typ′)))(dt_integral(typ′)‘from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_non_bool_integral_enum_TCC4.2.1:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral?(typ′)
{-3}	const?(typ′)
{1}	bottom?(dt_integral(typ′)‘from_byte(x'_1))
{2}	range(typ(typ′))(down(dt_integral(typ′)‘from_byte(x'_1)))

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.2.1`.
`dt_non_bool_integral_enum_TCC4.2.2:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral?(typ′)
{-3}	volatile?(typ′)
{1}	bottom?(dt_integral(typ′)‘from_byte(x'_1))
{2}	range(typ(typ′))(down(dt_integral(typ′)‘from_byte(x'_1)))

Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.2.2`.
`dt_non_bool_integral_enum_TCC4.3:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral?(typ′)
{1}	enum?(typ′)
{2}	∀ (x: int): range_integral(typ′)(x) ≡ range(typ′)(x)

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Keeping (-3 2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_TCC4.3`.
Q.E.D.

C.44.32 Cpp_Types.dt_non_bool_integral_enum_pod

Terse proof for `dt_non_bool_integral_enum_pod`.
`dt_non_bool_integral_enum_pod:`

{1}	∀ (typ: Cpp_Subtype(non_bool_integral_enum?)): pod_data_type?[(range(typ))](dt_non_bool_integral_enum(typ))

Repeatedly Skolemizing and flattening,
Expanding the definition of dt_non_bool_integral_enum,
Expanding the definition of non_bool_integral_enum?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`dt_non_bool_integral_enum_pod.1:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral?(typ′)
{-3}	enum?(typ′)
{1}	pod_data_type?[((range(typ′)))](dt_enum(typ′))

Keeping (-2 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_pod.1`.

`dt_non_bool_integral_enum_pod.2:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral?(typ′)
{1}	enum?(typ′)
{2}	pod_data_type?[((range(typ′)))](dt_integral(typ′))

Adding type constraints for dt_integral(typ!1),

Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_non_bool_integral_enum_pod.2.1:`

{-1}	uninterpreted_data_type?(dt_integral(typ′)‘uidt)
{-2}	∀ (*d*: ((range_integral(typ′))), *a*: Address): valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(*d*, *a*), *a*)
{-3}	∀ (*l*: list[Byte], *a*: Address): valid?(uidt(dt_integral(typ′)))(*l*, *a*) ≡ up?(from_byte(dt_integral(typ′))(*l*, *a*))
{-4}	∀ (*d*: ((range_integral(typ′))), *a*: Address): down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(*d*, *a*), *a*)) = *d*
{-5}	∀ (*l*: list[Byte], a_1, a_2: Address): valid?(uidt(dt_integral(typ′)))(*l*, a_1) ≡ valid?(uidt(dt_integral(typ′)))(*l*, a_2)
{-6}	size(uidt(dt_integral(typ′))) > 0
{-7}	Cpp_Type?(typ′)
{-8}	non_bool_integral?(typ′)
{1}	∀ (*d*: ((range(typ′))), *a*: Address): valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(*d*, *a*), *a*)
{2}	enum?(typ′)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_non_bool_integral_enum_pod.2.1.1:`

{-1} const?(typ′)
{-2} range(typ(typ′))(d′)
{-3} uninterpreted_data_type?(dt_integral(typ′)`uidt)
{-4} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(typ′)))(l, a) ≡
up?(from_byte(dt_integral(typ′))(l, a))
{-5} ∀ (d: ((range_integral(typ′))), a: Address):
down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(d, a), a)) = d
{-6} ∀ (l: list[Byte], a1, a2: Address):
valid?(uidt(dt_integral(typ′)))(l, a1) ≡
valid?(uidt(dt_integral(typ′)))(l, a2)
{-7} size(uidt(dt_integral(typ′))) > 0
{-8} Cpp_Type?(typ′)
{-9} non_bool_integral?(typ′)

{1} range_integral(typ′)(d′)
{2} valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(d′, a′), a′)

Keeping (-1 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_pod.2.1.1`.

`dt_non_bool_integral_enum_pod.2.1.2:`

{-1} range(typ(typ′))(d′)
{-2} volatile?(typ′)
{-3} uninterpreted_data_type?(dt_integral(typ′)`uidt)
{-4} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(typ′)))(l, a) ≡
up?(from_byte(dt_integral(typ′))(l, a))
{-5} ∀ (d: ((range_integral(typ′))), a: Address):
down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(d, a), a)) = d
{-6} ∀ (l: list[Byte], a1, a2: Address):
valid?(uidt(dt_integral(typ′)))(l, a1) ≡
valid?(uidt(dt_integral(typ′)))(l, a2)
{-7} size(uidt(dt_integral(typ′))) > 0
{-8} Cpp_Type?(typ′)
{-9} non_bool_integral?(typ′)

{1} range_integral(typ′)(d′)
{2} valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(d′, a′), a′)

Keeping (-2 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_pod.2.1.2`.

`dt_non_bool_integral_enum_pod.2.2:`

```
{-1}  uninterpreted_data_type?(dt_integral(typ')`uidt)
{-2}  ∀ (d: ((range_integral(typ'))), a: Address):
        valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_integral(typ')))(l, a) ≡
          up?(from_byte(dt_integral(typ'))(l, a))
{-4}  ∀ (d: ((range_integral(typ'))), a: Address):
        down(from_byte(dt_integral(typ'))(to_byte(dt_integral(typ'))(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_integral(typ')))(l, a1) ≡
          valid?(uidt(dt_integral(typ')))(l, a2)
{-6}  size(uidt(dt_integral(typ'))) > 0
{-7}  Cpp_Type?(typ')
{-8}  non_bool_integral?(typ')
------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        down(from_byte(dt_integral(typ'))(to_byte(dt_integral(typ'))(d, a), a)) = d
{2}   enum?(typ')
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_non_bool_integral_enum_pod.2.2.1:`

```
{-1}  const?(typ')
{-2}  range(typ(typ'))(d')
{-3}  uninterpreted_data_type?(dt_integral(typ')`uidt)
{-4}  ∀ (d: ((range_integral(typ'))), a: Address):
        valid?(uidt(dt_integral(typ')))(to_byte(dt_integral(typ'))(d, a), a)
{-5}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_integral(typ')))(l, a) ≡
          up?(from_byte(dt_integral(typ'))(l, a))
{-6}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_integral(typ')))(l, a1) ≡
          valid?(uidt(dt_integral(typ')))(l, a2)
{-7}  size(uidt(dt_integral(typ'))) > 0
{-8}  Cpp_Type?(typ')
{-9}  non_bool_integral?(typ')
------------------------------------------------------------
{1}   range_integral(typ')(d')
{2}   down(from_byte(dt_integral(typ'))(to_byte(dt_integral(typ'))(d', a'), a')) = d'
```

Keeping (-1 -9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_non_bool_integral_enum_pod.2.2.1`.

`dt_non_bool_integral_enum_pod.2.2.2:`

{-1} range(typ(typ′))(d′)
{-2} volatile?(typ′)
{-3} uninterpreted_data_type?(dt_integral(typ′)`uidt)
{-4} ∀ (d: ((range_integral(typ′))), a: Address):
valid?(uidt(dt_integral(typ′)))(to_byte(dt_integral(typ′))(d, a), a)
{-5} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(typ′)))(l, a) ≡
up?(from_byte(dt_integral(typ′))(l, a))
{-6} ∀ (l: list[Byte], a1, a2: Address):
valid?(uidt(dt_integral(typ′)))(l, a1) ≡
valid?(uidt(dt_integral(typ′)))(l, a2)
{-7} size(uidt(dt_integral(typ′))) > 0
{-8} Cpp_Type?(typ′)
{-9} non_bool_integral?(typ′)

{1} range_integral(typ′)(d′)
{2} down(from_byte(dt_integral(typ′))(to_byte(dt_integral(typ′))(d′, a′), a′)) = d′

Keeping (-2 -9) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_non_bool_integral_enum_pod.2.2.2`.
`dt_non_bool_integral_enum_pod.3:`

{-1} Cpp_Type?(typ′)
{-2} enum?(typ′)

{1} pod_data_type?[((range(typ′)))](dt_enum(typ′))

Adding type constraints for dt_enum(typ!1),
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_non_bool_integral_enum_pod.3.1:`

{-1} uninterpreted_data_type?(dt_enum(typ′)`uidt)
{-2} ∀ (d: ((range_enum(typ′))), a: Address):
valid?(uidt(dt_enum(typ′)))(to_byte(dt_enum(typ′))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_enum(typ′)))(l, a) ≡ up?(from_byte(dt_enum(typ′))(l, a))
{-4} ∀ (d: ((range_enum(typ′))), a: Address):
down(from_byte(dt_enum(typ′))(to_byte(dt_enum(typ′))(d, a), a)) = d
{-5} ∀ (l: list[Byte], a1, a2: Address):
valid?(uidt(dt_enum(typ′)))(l, a1) ≡ valid?(uidt(dt_enum(typ′)))(l, a2)
{-6} size(uidt(dt_enum(typ′))) > 0
{-7} Cpp_Type?(typ′)
{-8} enum?(typ′)

{1} ∀ (d: ((range(typ′))), a: Address):
valid?(uidt(dt_enum(typ′)))(to_byte(dt_enum(typ′))(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_non_bool_integral_enum_pod.3.1`.

`dt_non_bool_integral_enum_pod.3.2:`

{-1} uninterpreted_data_type?(dt_enum(typ′)‘uidt)
{-2} ∀ (d: ((range_enum(typ′))), a: Address):
valid?(uidt(dt_enum(typ′)))(to_byte(dt_enum(typ′))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_enum(typ′)))(l, a) ≡ up?(from_byte(dt_enum(typ′))(l, a))
{-4} ∀ (d: ((range_enum(typ′))), a: Address):
down(from_byte(dt_enum(typ′))(to_byte(dt_enum(typ′))(d, a), a)) = d
{-5} ∀ (l: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_enum(typ′)))(l, a_1) ≡ valid?(uidt(dt_enum(typ′)))(l, a_2)
{-6} size(uidt(dt_enum(typ′))) > 0
{-7} Cpp_Type?(typ′)
{-8} enum?(typ′)

{1} ∀ (d: ((range(typ′))), a: Address):
down(from_byte(dt_enum(typ′))(to_byte(dt_enum(typ′))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_non_bool_integral_enum_pod.3.2`.
Q.E.D.

C.44.33 Cpp_Types.dt_TCC1

Terse proof for `dt_TCC1`.

`dt_TCC1:`

{1} ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), t: Cpp_Type_):
typ = const(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range(t)))])): TRUE)

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of v,
Expanding the definition of c,
Replacing using formula -3,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`dt_TCC1.1:`

{-1} Cpp_Type?(const(t′))
{-2} non_bool_integral_enum?(const(t′))
{-3} typ′ = const(t′)

{1} ∃ (x_1: (interpreted_data_type?[((range(t′)))])): TRUE

Keeping (-2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.1`.

`dt_TCC1.2:`

{-1}	Cpp_Type?(const(t'))
{-2}	non_bool_integral_enum?(t')
{-3}	typ$'$ = const(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Expanding the definition of non_bool_integral_enum?,
Applying propositional simplification,
we get 2 subgoals:
`dt_TCC1.2.1:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Instantiating the top quantifier in 1 with the terms: (dt_integral(t!1)),
we get 3 subgoals:
`dt_TCC1.2.1.1:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	TRUE

which is trivially true.
This completes the proof of `dt_TCC1.2.1.1`.
`dt_TCC1.2.1.2:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x_1: [list[Byte], Address]): every[((range_integral(t')))] ((range(t')))(dt_integral(t')`from_byte(x_1))) $\wedge$ interpreted_data_type?[((range(t')))](dt_integral(t'))

Adding type constraints for dt_integral(t!1),
we get 2 subgoals:
`dt_TCC1.2.1.2.1:`

{-1}	pod_data_type?[((range_integral(t')))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ$'$ = const(t')
{1}	($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x_1: [list[Byte], Address]): every[((range_integral(t')))] ((range(t')))(dt_integral(t')`from_byte(x_1))) $\wedge$ interpreted_data_type?[((range(t')))](dt_integral(t'))

Applying propositional simplification,
we get 4 subgoals:

`dt_TCC1.2.1.2.1.1:`

{-1}	pod_data_type?[((range_integral(t')))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ$'$ = const(t')
{1}	$\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.2.1.2.1.1`.
`dt_TCC1.2.1.2.1.2:`

{-1}	pod_data_type?[((range_integral(t')))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ$'$ = const(t')
{1}	$\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.2.1.2.1.2`.
`dt_TCC1.2.1.2.1.3:`

{-1}	pod_data_type?[((range_integral(t')))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ$'$ = const(t')
{1}	$\forall$ (x_1: [list[Byte], Address]): every[((range_integral(t')))]((range(t')))(dt_integral(t')`from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range,
Keeping (-2 2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.2.1.2.1.3`.
`dt_TCC1.2.1.2.1.4:`

{-1}	pod_data_type?[((range_integral(t')))](dt_integral(t'))
{-2}	non_bool_integral?(t')
{-3}	Cpp_Type?(const(t'))
{-4}	typ$'$ = const(t')
{1}	interpreted_data_type?[((range(t')))](dt_integral(t'))

Using lemma pod_is_interpreted_data[(range_integral(t'))],
Keeping (-1 -3 1) and hiding *,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_TCC1.2.1.2.1.4.1:`

{-1} uninterpreted_data_type?(dt_integral(t')`uidt)
{-2} $\forall$ (d: ((range_integral(t'))), a: Address):
valid?(uidt(dt_integral(t')))(to_byte(dt_integral(t'))(d, a), a)
{-3} $\forall$ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(t')))(l, a) $\equiv$ up?(from_byte(dt_integral(t'))(l, a))
{-4} $\forall$ (d: ((range_integral(t'))), a: Address):
down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5} non_bool_integral?(t')

{1} $\forall$ (d: ((range(t'))), a: Address):
valid?(uidt(dt_integral(t')))(to_byte(dt_integral(t'))(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_TCC1.2.1.2.1.4.1`.

`dt_TCC1.2.1.2.1.4.2:`

{-1} uninterpreted_data_type?(dt_integral(t')`uidt)
{-2} $\forall$ (d: ((range_integral(t'))), a: Address):
valid?(uidt(dt_integral(t')))(to_byte(dt_integral(t'))(d, a), a)
{-3} $\forall$ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(t')))(l, a) $\equiv$ up?(from_byte(dt_integral(t'))(l, a))
{-4} $\forall$ (d: ((range_integral(t'))), a: Address):
down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5} non_bool_integral?(t')

{1} $\forall$ (d: ((range(t'))), a: Address):
down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_TCC1.2.1.2.1.4.2`.

`dt_TCC1.2.1.2.2:`

{-1} non_bool_integral?(t')
{-2} Cpp_Type?(const(t'))
{-3} typ$'$ = const(t')

{1} non_bool_integral?(t')
{2} ($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$
($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$
($\forall$ (x_1: [list[Byte], Address]):
every[((range_integral(t')))]
((range(t')))(dt_integral(t')`from_byte(x_1)))
$\wedge$ interpreted_data_type?[((range(t')))](dt_integral(t'))

which is trivially true.

This completes the proof of `dt_TCC1.2.1.2.2`.

`dt_TCC1.2.1.3:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	non_bool_integral?(t')

which is trivially true.
This completes the proof of `dt_TCC1.2.1.3`.
`dt_TCC1.2.2:`

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Instantiating the top quantifier in 1 with the terms: (dt_enum(t!1)),
we get 3 subgoals:
`dt_TCC1.2.2.1:`

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	TRUE

which is trivially true.
This completes the proof of `dt_TCC1.2.2.1`.
`dt_TCC1.2.2.2:`

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	($\forall$ (x: int): range_enum(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x: int): range_enum(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x_1: [list[Byte], Address]): every[((range_enum(t')))]((range(t')))(dt_enum(t')`from_byte(x_1))) $\wedge$ interpreted_data_type?[((range(t')))](dt_enum(t'))

Expanding the definition of range,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_TCC1.2.2.2.1:`

{-1}	enum?(t')
{-2}	Cpp_Type?(const(t'))
{-3}	typ$'$ = const(t')
{1}	interpreted_data_type?[((range(t')))](dt_enum(t'))

Adding type constraints for dt_enum(t!1),
Using lemma pod_is_interpreted_data[(range_enum(t'))],
Keeping (-1 -3 1) and hiding *,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_TCC1.2.2.2.1.1:`

{-1} uninterpreted_data_type?(dt_enum(t')`uidt)
{-2} $\forall$ (d: ((range_enum(t'))), a: Address):
valid?(uidt(dt_enum(t')))(to_byte(dt_enum(t'))(d, a), a)
{-3} $\forall$ (l: list[Byte], a: Address):
valid?(uidt(dt_enum(t')))(l, a) $\equiv$ up?(from_byte(dt_enum(t'))(l, a))
{-4} $\forall$ (d: ((range_enum(t'))), a: Address):
down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5} enum?(t')

{1} $\forall$ (d: ((range(t'))), a: Address):
valid?(uidt(dt_enum(t')))(to_byte(dt_enum(t'))(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_TCC1.2.2.2.1.1`.
`dt_TCC1.2.2.2.1.2:`

{-1} uninterpreted_data_type?(dt_enum(t')`uidt)
{-2} $\forall$ (d: ((range_enum(t'))), a: Address):
valid?(uidt(dt_enum(t')))(to_byte(dt_enum(t'))(d, a), a)
{-3} $\forall$ (l: list[Byte], a: Address):
valid?(uidt(dt_enum(t')))(l, a) $\equiv$ up?(from_byte(dt_enum(t'))(l, a))
{-4} $\forall$ (d: ((range_enum(t'))), a: Address):
down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5} enum?(t')

{1} $\forall$ (d: ((range(t'))), a: Address):
down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_TCC1.2.2.2.1.2`.
`dt_TCC1.2.2.2.2:`

{-1} enum?(t')
{-2} Cpp_Type?(const(t'))
{-3} typ$'$ = const(t')

{1} $\forall$ (x_1: [list[Byte], Address]):
every[((range_enum(t')))](range_enum(t'))(dt_enum(t')`from_byte(x_1))

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_TCC1.2.2.2.2`.
`dt_TCC1.2.2.3:`

{-1} enum?(t')
{-2} Cpp_Type?(const(t'))
{-3} typ$'$ = const(t')

{1} Cpp_Type?(t') $\wedge$ enum?(t')

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -3 with the terms: (t!2),
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_TCC1.2.2.3`.
`dt_TCC1.3:`

{-1}	Cpp_Type?(const(t'))
{-2}	volatile?(t')
{-3}	non_bool_integral_enum?(typ(t'))
{-4}	typ$'$ = const(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Expanding the definition of non_bool_integral_enum?,
Applying propositional simplification,
we get 2 subgoals:
`dt_TCC1.3.1:`

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ$'$ = const(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Instantiating the top quantifier in 1 with the terms: (dt_integral(typ(t!1))),
we get 3 subgoals:
`dt_TCC1.3.1.1:`

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ$'$ = const(t')
{1}	TRUE

which is trivially true.
This completes the proof of `dt_TCC1.3.1.1`.
`dt_TCC1.3.1.2:`

{-1}	non_bool_integral?(typ(t'))
{-2}	Cpp_Type?(const(t'))
{-3}	volatile?(t')
{-4}	typ$'$ = const(t')
{1}	($\forall$ (x: int): range_integral(typ(t'))(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x: int): range_integral(typ(t'))(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x_1: [list[Byte], Address]): every[((range_integral(typ(t'))))] ((range(t')))(dt_integral(typ(t'))‘from_byte(x_1))) $\wedge$ interpreted_data_type?[((range(t')))](dt_integral(typ(t')))

Adding type constraints for dt_integral(typ(t!1)),
we get 2 subgoals:

`dt_TCC1.3.1.2.1:`

```
{-1}  pod_data_type?[((range_integral(typ(t'))))](dt_integral(typ(t')))
{-2}  non_bool_integral?(typ(t'))
{-3}  Cpp_Type?(const(t'))
{-4}  volatile?(t')
{-5}  typ' = const(t')
------------------------------------------------------------------------
{1}   (∀ (x: int): range_integral(typ(t'))(x) ≡ range(t')(x)) ∧
       (∀ (x: int): range_integral(typ(t'))(x) ≡ range(t')(x)) ∧
        (∀ (x1: [list[Byte], Address]):
           every[((range_integral(typ(t'))))]
                ((range(t')))(dt_integral(typ(t'))`from_byte(x1)))
          ∧ interpreted_data_type?[((range(t')))](dt_integral(typ(t')))
```

Applying propositional simplification,
we get 4 subgoals:
`dt_TCC1.3.1.2.1.1:`

```
{-1}  pod_data_type?[((range_integral(typ(t'))))](dt_integral(typ(t')))
{-2}  non_bool_integral?(typ(t'))
{-3}  Cpp_Type?(const(t'))
{-4}  volatile?(t')
{-5}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (x: int): range_integral(typ(t'))(x) ≡ range(t')(x)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.3.1.2.1.1`.
`dt_TCC1.3.1.2.1.2:`

```
{-1}  pod_data_type?[((range_integral(typ(t'))))](dt_integral(typ(t')))
{-2}  non_bool_integral?(typ(t'))
{-3}  Cpp_Type?(const(t'))
{-4}  volatile?(t')
{-5}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (x: int): range_integral(typ(t'))(x) ≡ range(t')(x)
```

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.3.1.2.1.2`.
`dt_TCC1.3.1.2.1.3:`

```
{-1}  pod_data_type?[((range_integral(typ(t'))))](dt_integral(typ(t')))
{-2}  non_bool_integral?(typ(t'))
{-3}  Cpp_Type?(const(t'))
{-4}  volatile?(t')
{-5}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[((range_integral(typ(t'))))]
             ((range(t')))(dt_integral(typ(t'))`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of range,

Keeping (-2 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_TCC1.3.1.2.1.3`.

`dt_TCC1.3.1.2.1.4:`

```
{-1}  pod_data_type?[((range_integral(typ(t'))))](dt_integral(typ(t')))
{-2}  non_bool_integral?(typ(t'))
{-3}  Cpp_Type?(const(t'))
{-4}  volatile?(t')
{-5}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range(t')))](dt_integral(typ(t')))
```

Using lemma pod_is_interpreted_data[(range_integral(typ(t')))],

Keeping (-1 -3 1) and hiding *,

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_TCC1.3.1.2.1.4.1:`

```
{-1}  uninterpreted_data_type?(dt_integral(typ(t'))`uidt)
{-2}  ∀ (d: ((range_integral(typ(t')))), a: Address):
        valid?(uidt(dt_integral(typ(t'))))(to_byte(dt_integral(typ(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_integral(typ(t'))))(l, a) ≡
          up?(from_byte(dt_integral(typ(t')))(l, a))
{-4}  ∀ (d: ((range_integral(typ(t')))), a: Address):
        down(from_byte(dt_integral(typ(t')))
                      (to_byte(dt_integral(typ(t')))(d, a), a))
          = d
{-5}  non_bool_integral?(typ(t'))
------------------------------------------------------------------------
{1}   ∀ (d: ((range(t'))), a: Address):
        valid?(uidt(dt_integral(typ(t'))))(to_byte(dt_integral(typ(t')))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Keeping (-1 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_TCC1.3.1.2.1.4.1`.

`dt_TCC1.3.1.2.1.4.2:`

{-1} uninterpreted_data_type?(dt_integral(typ(t'))`uidt)
{-2} ∀ (d: ((range_integral(typ(t')))), a: Address):
valid?(uidt(dt_integral(typ(t'))))(to_byte(dt_integral(typ(t')))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_integral(typ(t'))))(l, a) ≡
up?(from_byte(dt_integral(typ(t')))(l, a))
{-4} ∀ (d: ((range_integral(typ(t')))), a: Address):
down(from_byte(dt_integral(typ(t')))
(to_byte(dt_integral(typ(t')))(d, a), a))
= d
{-5} non_bool_integral?(typ(t'))

{1} ∀ (d: ((range(t'))), a: Address):
down(from_byte(dt_integral(typ(t')))
(to_byte(dt_integral(typ(t')))(d, a), a))
= d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC1.3.1.2.1.4.2`.

`dt_TCC1.3.1.2.2:`

{-1} non_bool_integral?(typ(t'))
{-2} Cpp_Type?(const(t'))
{-3} volatile?(t')
{-4} typ′ = const(t')

{1} non_bool_integral?(typ(t'))
{2} (∀ (x: int): range_integral(typ(t'))(x) ≡ range(t')(x)) ∧
(∀ (x: int): range_integral(typ(t'))(x) ≡ range(t')(x)) ∧
(∀ (x_1: [list[Byte], Address]):
every[((range_integral(typ(t'))))]
((range(t')))(dt_integral(typ(t'))`from_byte(x_1)))
∧ interpreted_data_type?[((range(t')))](dt_integral(typ(t')))

which is trivially true.
This completes the proof of `dt_TCC1.3.1.2.2`.

`dt_TCC1.3.1.3:`

{-1} non_bool_integral?(typ(t'))
{-2} Cpp_Type?(const(t'))
{-3} volatile?(t')
{-4} typ′ = const(t')

{1} non_bool_integral?(typ(t'))

which is trivially true.
This completes the proof of `dt_TCC1.3.1.3`.

`dt_TCC1.3.2:`

{-1} enum?(typ(t'))
{-2} Cpp_Type?(const(t'))
{-3} volatile?(t')
{-4} typ′ = const(t')

{1} ∃ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Instantiating the top quantifier in 1 with the terms: (dt_enum(typ(t!1))),
we get 3 subgoals:
`dt_TCC1.3.2.1:`

{-1} enum?(typ(t'))
{-2} Cpp_Type?(const(t'))
{-3} volatile?(t')
{-4} typ$'$ = const(t')

{1} TRUE

which is trivially true.
This completes the proof of `dt_TCC1.3.2.1`.
`dt_TCC1.3.2.2:`

{-1} enum?(typ(t'))
{-2} Cpp_Type?(const(t'))
{-3} volatile?(t')
{-4} typ$'$ = const(t')

{1} (∀ (x: int): range_enum(typ(t'))(x) ≡ range(t')(x)) ∧
(∀ (x: int): range_enum(typ(t'))(x) ≡ range(t')(x)) ∧
(∀ (x_1: [list[Byte], Address]):
every[((range_enum(typ(t'))))]
((range(t')))(dt_enum(typ(t'))‘from_byte(x_1)))
∧ interpreted_data_type?[((range(t')))](dt_enum(typ(t')))

Expanding the definition of range,
Expanding the definition of range,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_TCC1.3.2.2.1:`

{-1} enum?(typ(t'))
{-2} Cpp_Type?(const(t'))
{-3} volatile?(t')
{-4} typ$'$ = const(t')

{1} interpreted_data_type?[((range(t')))](dt_enum(typ(t')))

Adding type constraints for dt_enum(typ(t!1)),
Using lemma pod_is_interpreted_data[(range_enum(typ(t')))],
Keeping (-1 -3 1) and hiding *,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_TCC1.3.2.2.1.1:`

{-1} uninterpreted_data_type?(dt_enum(typ(t'))‘uidt)
{-2} ∀ (d: ((range_enum(typ(t')))), a: Address):
valid?(uidt(dt_enum(typ(t'))))(to_byte(dt_enum(typ(t')))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_enum(typ(t'))))(l, a) ≡
up?(from_byte(dt_enum(typ(t')))(l, a))
{-4} ∀ (d: ((range_enum(typ(t')))), a: Address):
down(from_byte(dt_enum(typ(t')))(to_byte(dt_enum(typ(t')))(d, a), a)) = d
{-5} enum?(typ(t'))

{1} ∀ (d: ((range(t'))), a: Address):
valid?(uidt(dt_enum(typ(t'))))(to_byte(dt_enum(typ(t')))(d, a), a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -5 1) and hiding *,
Expanding the definition of range,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_TCC1.3.2.2.1.1`.
`dt_TCC1.3.2.2.1.2:`

```
{-1}  uninterpreted_data_type?(dt_enum(typ(t'))`uidt)
{-2}  ∀ (d: ((range_enum(typ(t')))), a: Address):
        valid?(uidt(dt_enum(typ(t'))))(to_byte(dt_enum(typ(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_enum(typ(t'))))(l, a) ≡
          up?(from_byte(dt_enum(typ(t')))(l, a))
{-4}  ∀ (d: ((range_enum(typ(t')))), a: Address):
        down(from_byte(dt_enum(typ(t')))(to_byte(dt_enum(typ(t')))(d, a), a)) = d
{-5}  enum?(typ(t'))
------------------------------------------------------------------------
 {1}  ∀ (d: ((range(t'))), a: Address):
        down(from_byte(dt_enum(typ(t')))(to_byte(dt_enum(typ(t')))(d, a), a)) = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_TCC1.3.2.2.1.2`.
`dt_TCC1.3.2.2.2:`

```
{-1}  enum?(typ(t'))
{-2}  Cpp_Type?(const(t'))
{-3}  volatile?(t')
{-4}  typ' = const(t')
------------------------------------------------------------------------
 {1}  ∀ (x1: [list[Byte], Address]):
        every[((range_enum(typ(t'))))]
             (range_enum(typ(t')))(dt_enum(typ(t'))`from_byte(x1))
```

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_TCC1.3.2.2.2`.
`dt_TCC1.3.2.3:`

```
{-1}  enum?(typ(t'))
{-2}  Cpp_Type?(const(t'))
{-3}  volatile?(t')
{-4}  typ' = const(t')
------------------------------------------------------------------------
 {1}  Cpp_Type?(typ(t')) ∧ enum?(typ(t'))
```

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -3 with the terms: (t!2),
Expanding the definition of subterm,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC1.3.2.3.
Q.E.D.

C.44.34 Cpp_Types.dt_TCC2

Terse proof for dt_TCC2.
dt_TCC2:

```
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
          v:
            [typ1:
                {z: Cpp_Subtype(cv(non_bool_integral_enum?)) |
                            restrict
                                [[Cpp_Type_, Cpp_Type_],
                                  [Cpp_Subtype(cv(non_bool_integral_enum?)),
                                    Cpp_Subtype(cv(non_bool_integral_enum?))],
                                  boolean]
                                (≪)(z, typ)} →
                    (interpreted_data_type?[((range(typ1)))])],
            t: Cpp_Type_):
       typ = const(t) ⊃
        (∀ (x: int): range(t)(x) ≡ range(typ)(x)) ∧
         (∀ (x: int): range(t)(x) ≡ range(typ)(x)) ∧
          (∀ (x1: [list[Byte], Address]):
              every[((range(t)))]
                  ((range(typ)))(dt_const[((range(t)))](v(t))‘from_byte(x1)))
           ∧ interpreted_data_type?[(range(typ))](dt_const[((range(t)))](v(t)))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of range,
Applying propositional simplification,
we get 2 subgoals:
dt_TCC2.1:

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(non_bool_integral_enum?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[((range(t')))]
            ((range(const(t'))))
            (dt_const[((range(t')))](v'(t'))‘from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range,
which is trivially true.
This completes the proof of dt_TCC2.1.

`dt_TCC2.2:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(non_bool_integral_enum?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range(typ')))](dt_const[((range(t')))](v'(t')))
```

Adding type constraints for dt_const(v!1(t!1)),

we get 2 subgoals:

`dt_TCC2.2.1:`

```
{-1}  interpreted_data_type?[((range(t')))](dt_const(v'(t')))
{-2}  Cpp_Type?(const(t'))
{-3}  cv(non_bool_integral_enum?)(const(t'))
{-4}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range(typ')))](dt_const[((range(t')))](v'(t')))
```

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_TCC2.2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_const(v'(t'))`uidt)
{-2}  ∀ (d: ((range(t'))), a: Address):
        valid?(uidt(dt_const(v'(t'))))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const(v'(t'))))(l, a) ≡
          up?(from_byte(dt_const(v'(t')))(l, a))
{-4}  ∀ (d: ((range(t'))), a: Address):
        down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)) =
          d
{-5}  Cpp_Type?(const(t'))
{-6}  cv(non_bool_integral_enum?)(const(t'))
{-7}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        valid?(uidt(dt_const[((range(t')))](v'(t'))))
            (to_byte(dt_const[((range(t')))](v'(t')))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_TCC2.2.1.1`.

`dt_TCC2.2.1.2`:

{-1}	uninterpreted_data_type?(dt_const($v'(t')$)`uidt)
{-2}	∀ (d: ((range(t'))), a: Address): valid?(uidt(dt_const($v'(t')$))(to_byte(dt_const($v'(t')$))(d, a), a)
{-3}	∀ (l: list[Byte], a: Address): valid?(uidt(dt_const($v'(t')$))(l, a) ≡ up?(from_byte(dt_const($v'(t')$))(l, a))
{-4}	∀ (d: ((range(t'))), a: Address): down(from_byte(dt_const($v'(t')$))(to_byte(dt_const($v'(t')$))(d, a), a)) = d
{-5}	Cpp_Type?(const(t'))
{-6}	cv(non_bool_integral_enum?)(const(t'))
{-7}	typ′ = const(t')
{1}	∀ (d: ((range(typ′))), a: Address): down(from_byte(dt_const[((range(t')))]($v'(t')$)) (to_byte(dt_const[((range(t')))]($v'(t')$))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_TCC2.2.1.2`.
`dt_TCC2.2.2`:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(non_bool_integral_enum?)(const(t'))
{-3}	typ′ = const(t')
{1}	∃ (x_1: (interpreted_data_type?[((range(t')))])): TRUE
{2}	interpreted_data_type?[((range(typ′)))](dt_const[((range(t')))]($v'(t')$))

Using lemma dt_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_TCC2.2.2`.
Q.E.D.

C.44.35 Cpp_Types.dt_TCC3

Terse proof for `dt_TCC3`.
`dt_TCC3`:

{1}	∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)), t: Cpp_Type_): typ = volatile(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range(t)))])): TRUE)

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`dt_TCC3.1:`

{-1}	Cpp_Type?(volatile(t'))
{-2}	non_bool_integral_enum?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Keeping (-2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC3.1`.
`dt_TCC3.2:`

{-1}	Cpp_Type?(volatile(t'))
{-2}	non_bool_integral_enum?(t')
{-3}	typ$'$ = volatile(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Expanding the definition of non_bool_integral_enum?,
Applying propositional simplification,
we get 2 subgoals:
`dt_TCC3.2.1:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Instantiating the top quantifier in 1 with the terms: (dt_integral(t!1)),
we get 3 subgoals:
`dt_TCC3.2.1.1:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	TRUE

which is trivially true.
This completes the proof of `dt_TCC3.2.1.1`.
`dt_TCC3.2.1.2:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)) $\wedge$ ($\forall$ (x_1: [list[Byte], Address]): every[((range_integral(t')))] ((range(t')))(dt_integral(t')`from_byte(x_1))) $\wedge$ interpreted_data_type?[((range(t')))](dt_integral(t'))

Applying propositional simplification,
we get 4 subgoals:
`dt_TCC3.2.1.2.1:`

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\forall$ (x: int): range_integral(t')(x) $\equiv$ range(t')(x)

Keeping (-1 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC3.2.1.2.1`.
`dt_TCC3.2.1.2.2:`

```
{-1}  non_bool_integral?(t')
{-2}  Cpp_Type?(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------
{1}   ∀ (x: int): range_integral(t')(x) ≡ range(t')(x)
```

Keeping (-1 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC3.2.1.2.2`.
`dt_TCC3.2.1.2.3:`

```
{-1}  non_bool_integral?(t')
{-2}  Cpp_Type?(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------
{1}   ∀ (x1: ⌊list⌊Byte⌋, Address⌋):
        every⌈((range_integral(t')))⌉((range(t')))(dt_integral(t')‘from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Adding type constraints for down(dt_integral(t!1)‘from_byte(x1!1)),
Expanding the definition of range,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC3.2.1.2.3`.
`dt_TCC3.2.1.2.4:`

```
{-1}  non_bool_integral?(t')
{-2}  Cpp_Type?(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------
{1}   interpreted_data_type?⌈((range(t')))⌉(dt_integral(t'))
```

Adding type constraints for dt_integral(t!1),
Using lemma pod_is_interpreted_data⌈(range_integral(t'))⌉,
Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_TCC3.2.1.2.4.1:`

```
{-1}  uninterpreted_data_type?(dt_integral(t')‘uidt)
{-2}  ∀ (d: ((range_integral(t'))), a: Address):
        valid?(uidt(dt_integral(t')))(to_byte(dt_integral(t'))(d, a), a)
{-3}  ∀ (l: list⌊Byte⌋, a: Address):
        valid?(uidt(dt_integral(t')))(l, a) ≡ up?(from_byte(dt_integral(t'))(l, a))
{-4}  ∀ (d: ((range_integral(t'))), a: Address):
        down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5}  pod_data_type?⌈((range_integral(t')))⌉(dt_integral(t'))
{-6}  non_bool_integral?(t')
{-7}  Cpp_Type?(volatile(t'))
{-8}  typ' = volatile(t')
------------------------------------------------------------
{1}   ∀ (d: ((range(t'))), a: Address):
        valid?(uidt(dt_integral(t')))(to_byte(dt_integral(t'))(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -6 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC3.2.1.2.4.1`.
`dt_TCC3.2.1.2.4.2`:

{-1}	uninterpreted_data_type?(dt_integral(t')`uidt)
{-2}	$\forall$ (d: ((range_integral(t'))), a: Address): valid?(uidt(dt_integral(t')))(to_byte(dt_integral(t'))(d, a), a)
{-3}	$\forall$ (l: list[Byte], a: Address): valid?(uidt(dt_integral(t')))(l, a) $\equiv$ up?(from_byte(dt_integral(t'))(l, a))
{-4}	$\forall$ (d: ((range_integral(t'))), a: Address): down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d
{-5}	pod_data_type?[((range_integral(t')))](dt_integral(t'))
{-6}	non_bool_integral?(t')
{-7}	Cpp_Type?(volatile(t'))
{-8}	typ$'$ = volatile(t')
{1}	$\forall$ (d: ((range(t'))), a: Address): down(from_byte(dt_integral(t'))(to_byte(dt_integral(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 -6 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_TCC3.2.1.2.4.2`.
`dt_TCC3.2.1.3`:

{-1}	non_bool_integral?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	non_bool_integral?(t')

which is trivially true.
This completes the proof of `dt_TCC3.2.1.3`.
`dt_TCC3.2.2`:

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range(t')))])): TRUE

Instantiating the top quantifier in 1 with the terms: (dt_enum(t!1)),
we get 3 subgoals:
`dt_TCC3.2.2.1`:

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	TRUE

which is trivially true.
This completes the proof of `dt_TCC3.2.2.1`.

`dt_TCC3.2.2.2:`

```
{-1}  enum?(t')
{-2}  Cpp_Type?(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   (∀ (x: int): range_enum(t')(x) ≡ range(t')(x)) ∧
        (∀ (x: int): range_enum(t')(x) ≡ range(t')(x)) ∧
          (∀ (x1: [list[Byte], Address]):
               every[((range_enum(t')))]((range(t')))(dt_enum(t')`from_byte(x1)))
            ∧ interpreted_data_type?[((range(t')))](dt_enum(t'))
```

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`dt_TCC3.2.2.2.1:`

```
{-1}  enum?(t')
{-2}  Cpp_Type?(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range(t')))](dt_enum(t'))
```

Adding type constraints for dt_enum(t!1),

Using lemma pod_is_interpreted_data[(range_enum(t'))],

Hiding formulas: (-2 -4 -5),

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_TCC3.2.2.2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_enum(t')`uidt)
{-2}  ∀ (d: ((range_enum(t'))), a: Address):
        valid?(uidt(dt_enum(t')))(to_byte(dt_enum(t'))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_enum(t')))(l, a) ≡ up?(from_byte(dt_enum(t'))(l, a))
{-4}  ∀ (d: ((range_enum(t'))), a: Address):
        down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5}  enum?(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range(t'))), a: Address):
        valid?(uidt(dt_enum(t')))(to_byte(dt_enum(t'))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_TCC3.2.2.2.1.1`.

`dt_TCC3.2.2.2.1.2:`

{-1}	uninterpreted_data_type?(dt_enum(t')`uidt)
{-2}	$\forall$ (d: ((range_enum(t'))), a: Address): valid?(uidt(dt_enum(t')))(to_byte(dt_enum(t'))(d, a), a)
{-3}	$\forall$ (l: list[Byte], a: Address): valid?(uidt(dt_enum(t')))(l, a) $\equiv$ up?(from_byte(dt_enum(t'))(l, a))
{-4}	$\forall$ (d: ((range_enum(t'))), a: Address): down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d
{-5}	enum?(t')
{1}	$\forall$ (d: ((range(t'))), a: Address): down(from_byte(dt_enum(t'))(to_byte(dt_enum(t'))(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_TCC3.2.2.2.1.2`.

`dt_TCC3.2.2.2.2:`

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\forall$ (x_1: [list[Byte], Address]): every[((range_enum(t')))](range_enum(t'))(dt_enum(t')`from_byte(x_1))

Expanding the definition of every,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_TCC3.2.2.2.2`.

`dt_TCC3.2.2.3:`

{-1}	enum?(t')
{-2}	Cpp_Type?(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	Cpp_Type?(t') $\wedge$ enum?(t')

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -3 with the terms: (t!2),

Expanding the definition of subterm,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_TCC3.2.2.3`.

Q.E.D.

C.44.36 Cpp_Types.dt_TCC4

Terse proof for `dt_TCC4`.

dt_TCC4:

```
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?)),
          v:
            [typ1:
               {z: Cpp_Subtype(cv(non_bool_integral_enum?)) |
                     restrict
                       [[Cpp_Type_, Cpp_Type_],
                         [Cpp_Subtype(cv(non_bool_integral_enum?)),
                           Cpp_Subtype(cv(non_bool_integral_enum?))],
                         boolean]
                       (≪)(z, typ)} →
                 (interpreted_data_type?[((range(typ1)))])],
            t: Cpp_Type_):
       typ = volatile(t) ⊃
        (∀ (x: int): range(t)(x) ≡ range(typ)(x)) ∧
         (∀ (x: int): range(t)(x) ≡ range(typ)(x)) ∧
          (∀ (x1: [list[Byte], Address]):
             every[((range(t)))]
               ((range(typ)))
               (dt_volatile[((range(t)))](v(t))`from_byte(x1)))
           ∧
           interpreted_data_type?[(range(typ))](dt_volatile[((range(t)))](v(t)))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of range,
Applying propositional simplification,
we get 2 subgoals:
dt_TCC4.1:

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(non_bool_integral_enum?)(volatile(t'))
{-3}  typ' = volatile(t')
{1}  ∀ (x1: [list[Byte], Address]):
       every[((range(t')))]
         ((range(volatile(t'))))
         (dt_volatile[((range(t')))](v'(t'))`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of range,
which is trivially true.
This completes the proof of dt_TCC4.1.
dt_TCC4.2:

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(non_bool_integral_enum?)(volatile(t'))
{-3}  typ' = volatile(t')
{1}  interpreted_data_type?[((range(typ')))]
       (dt_volatile[((range(t')))](v'(t')))
```

Adding type constraints for dt_volatile(v!1(t!1)),

we get 2 subgoals:

dt_TCC4.2.1:

```
{-1}  interpreted_data_type?[((range(t')))](dt_volatile(v'(t')))
{-2}  Cpp_Type?(volatile(t'))
{-3}  cv(non_bool_integral_enum?)(volatile(t'))
{-4}  typ' = volatile(t')
------------------------------------------------------------------
{1}   interpreted_data_type?[((range(typ')))]
        (dt_volatile[((range(t')))](v'(t')))
```

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

dt_TCC4.2.1.1:

```
{-1}  uninterpreted_data_type?(dt_volatile(v'(t'))`uidt)
{-2}  ∀ (d: ((range(t'))), a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(to_byte(dt_volatile(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(l, a) ≡
          up?(from_byte(dt_volatile(v'(t')))(l, a))
{-4}  ∀ (d: ((range(t'))), a: Address):
        down(from_byte(dt_volatile(v'(t')))
                      (to_byte(dt_volatile(v'(t')))(d, a), a))
          = d
{-5}  Cpp_Type?(volatile(t'))
{-6}  cv(non_bool_integral_enum?)(volatile(t'))
{-7}  typ' = volatile(t')
------------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        valid?(uidt(dt_volatile[((range(t')))](v'(t'))))
          (to_byte(dt_volatile[((range(t')))](v'(t')))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_TCC4.2.1.1.

dt_TCC4.2.1.2:

```
{-1}  uninterpreted_data_type?(dt_volatile(v'(t'))`uidt)
{-2}  ∀ (d: ((range(t'))), a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(to_byte(dt_volatile(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(l, a) ≡
          up?(from_byte(dt_volatile(v'(t')))(l, a))
{-4}  ∀ (d: ((range(t'))), a: Address):
        down(from_byte(dt_volatile(v'(t')))
                      (to_byte(dt_volatile(v'(t')))(d, a), a))
          = d
{-5}  Cpp_Type?(volatile(t'))
{-6}  cv(non_bool_integral_enum?)(volatile(t'))
{-7}  typ' = volatile(t')
-------------------------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        down(from_byte(dt_volatile[((range(t')))](v'(t')))
                      (to_byte(dt_volatile[((range(t')))](v'(t')))(d, a), a))
          = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of range,
which is trivially true.
This completes the proof of dt_TCC4.2.1.2.
dt_TCC4.2.2:

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(non_bool_integral_enum?)(volatile(t'))
{-3}  typ' = volatile(t')
-------------------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[((range(t')))])): TRUE
{2}   interpreted_data_type?[((range(typ')))]
          (dt_volatile[((range(t')))](v'(t')))
```

Using lemma dt_TCC3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_TCC4.2.2.
Q.E.D.

C.44.37 Cpp_Types.dt_TCC5

Terse proof for dt_TCC5.
dt_TCC5:

```
-------------------------------------------------------------------------------
{1}   ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
        ¬ volatile?(typ) ∧ ¬ const?(typ) ⊃ non_bool_integral_enum?(typ)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_TCC5.
Q.E.D.

C.44.38 Cpp_Types.dt_pod

Terse proof for dt_pod.

`dt_pod:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
       pod_data_type?[(range(typ))](dt(typ))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: cv c v
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`dt_pod.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  non_bool_integral_enum?(typ′)
{1}   pod_data_type?[((range(typ′)))](dt(typ′))
```

Expanding the definition of dt,
Adding type constraints for dt_non_bool_integral_enum(typ!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_pod.1.1:`

```
{-1}  pod_data_type?[((range(typ′)))](dt_non_bool_integral_enum(typ′))
{-2}  Cpp_Type?(typ′)
{-3}  non_bool_integral_enum?(typ′)
{-4}  const?(typ′)
{1}   pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))
```

Keeping (-3 -4) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.1.1`.
`dt_pod.1.2:`

```
{-1}  pod_data_type?[((range(typ′)))](dt_non_bool_integral_enum(typ′))
{-2}  Cpp_Type?(typ′)
{-3}  non_bool_integral_enum?(typ′)
{-4}  volatile?(typ′)
{1}   pod_data_type?[((range(typ′)))](dt_volatile(dt(typ(typ′))))
```

Keeping (-3 -4) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.1.2`.
`dt_pod.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  const?(typ′)
{-3}  non_bool_integral_enum?(typ(typ′))
{1}   pod_data_type?[((range(typ′)))](dt(typ′))
```

Expanding the definition of dt,
Using lemma dt_const_pod[(range(typ(typ′)))],
we get 3 subgoals:

`dt_pod.2.1:`

{-1} pod_data_type?[(range(typ(typ′)))](dt_const(dt(typ(typ′))))
{-2} Cpp_Type?(typ′)
{-3} const?(typ′)
{-4} non_bool_integral_enum?(typ(typ′))

{1} pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))

Expanding the definition of pod_data_type?,
Adding type constraints for dt_const(dt(typ(typ!1))),
we get 2 subgoals:
`dt_pod.2.1.1:`

{-1} interpreted_data_type?[(range(typ(typ′)))](dt_const(dt(typ(typ′))))
{-2} (∀ (*l*: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_1) ≡
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_2))
∧ size(uidt(dt_const(dt(typ(typ′))))) > 0
{-3} Cpp_Type?(typ′)
{-4} const?(typ′)
{-5} non_bool_integral_enum?(typ(typ′))

{1} interpreted_data_type?(dt_const(dt(typ(typ′)))) ∧
(∀ (*l*: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_1) ≡
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_2))
∧ size(uidt(dt_const(dt(typ(typ′))))) > 0

Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_pod.2.1.1.1:`

{-1} uninterpreted_data_type?(dt_const(dt(typ(typ′)))‘uidt)
{-2} ∀ (*d*: ((range(typ(typ′)))), *a*: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))
(to_byte(dt_const(dt(typ(typ′))))(*d*, *a*), *a*)
{-3} ∀ (*l*: list[Byte], *a*: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, *a*) ≡
up?(from_byte(dt_const(dt(typ(typ′))))(*l*, *a*))
{-4} ∀ (*d*: ((range(typ(typ′)))), *a*: Address):
down(from_byte(dt_const(dt(typ(typ′))))
(to_byte(dt_const(dt(typ(typ′))))(*d*, *a*), *a*))
= *d*
{-5} ∀ (*l*: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_1) ≡
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_2)
{-6} size(uidt(dt_const(dt(typ(typ′))))) > 0
{-7} Cpp_Type?(typ′)
{-8} const?(typ′)
{-9} non_bool_integral_enum?(typ(typ′))

{1} ∀ (*d*: ((range(typ′))), *a*: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))
(to_byte(dt_const(dt(typ(typ′))))(*d*, *a*), *a*)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_pod.2.1.1.1`.

`dt_pod.2.1.1.2`:

```
{-1}  uninterpreted_data_type?(dt_const(dt(typ(typ′)))`uidt)
{-2}  ∀ (d: ((range(typ(typ′)))), a: Address):
        valid?(uidt(dt_const(dt(typ(typ′)))))
             (to_byte(dt_const(dt(typ(typ′))))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const(dt(typ(typ′)))))(l, a) ≡
         up?(from_byte(dt_const(dt(typ(typ′))))(l, a))
{-4}  ∀ (d: ((range(typ(typ′)))), a: Address):
        down(from_byte(dt_const(dt(typ(typ′))))
                      (to_byte(dt_const(dt(typ(typ′))))(d, a), a))
         = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_const(dt(typ(typ′)))))(l, a1) ≡
         valid?(uidt(dt_const(dt(typ(typ′)))))(l, a2)
{-6}  size(uidt(dt_const(dt(typ(typ′))))) > 0
{-7}  Cpp_Type?(typ′)
{-8}  const?(typ′)
{-9}  non_bool_integral_enum?(typ(typ′))
-------------------------------------------------------------
{1}   ∀ (d: ((range(typ′))), a: Address):
        down(from_byte(dt_const(dt(typ(typ′))))
                      (to_byte(dt_const(dt(typ(typ′))))(d, a), a))
         = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_pod.2.1.1.2`.

`dt_pod.2.1.2`:

```
{-1}  (∀ (l: list[Byte], a1, a2: Address):
           valid?(uidt(dt_const(dt(typ(typ′)))))(l, a1) ≡
            valid?(uidt(dt_const(dt(typ(typ′)))))(l, a2))
       ∧ size(uidt(dt_const(dt(typ(typ′))))) > 0
{-2}  Cpp_Type?(typ′)
{-3}  const?(typ′)
{-4}  non_bool_integral_enum?(typ(typ′))
-------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?⌊(range(typ(typ′)))⌋)): TRUE
{2}   interpreted_data_type?(dt_const(dt(typ(typ′)))) ∧
       (∀ (l: list[Byte], a1, a2: Address):
           valid?(uidt(dt_const(dt(typ(typ′)))))(l, a1) ≡
            valid?(uidt(dt_const(dt(typ(typ′)))))(l, a2))
        ∧ size(uidt(dt_const(dt(typ(typ′))))) > 0
```

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),

which is trivially true.

This completes the proof of `dt_pod.2.1.2`.

`dt_pod.2.2:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{1}	pod_data_type?[(range(typ(typ′)))](dt(typ(typ′)))
{2}	pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))

Hiding formulas: (-2 2),
Expanding the definition of dt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_pod.2.2.1:`

{-1}	const?(typ(typ′))
{-2}	Cpp_Type?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{1}	pod_data_type?[(range(typ(typ′)))](dt_const(dt(typ(typ(typ′)))))

Keeping (-1 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.2.2.1`.
`dt_pod.2.2.2:`

{-1}	volatile?(typ(typ′))
{-2}	Cpp_Type?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{1}	pod_data_type?[(range(typ(typ′)))](dt_volatile(dt(typ(typ(typ′)))))

Keeping (-1 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.2.2.2`.
`dt_pod.2.3:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{1}	∃ (x_1: (interpreted_data_type?[(range(typ(typ′)))])): TRUE
{2}	pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),
which is trivially true.
This completes the proof of `dt_pod.2.3`.
`dt_pod.3:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	volatile?(typ(typ′))
{-4}	non_bool_integral_enum?(typ(typ(typ′)))
{1}	pod_data_type?[((range(typ′)))](dt(typ′))

Expanding the definition of dt,
Using lemma dt_const_pod[(range(typ(typ′)))],
we get 3 subgoals:

`dt_pod.3.1:`

{-1} pod_data_type?[(range(typ(typ′)))](dt_const(dt(typ(typ′))))
{-2} Cpp_Type?(typ′)
{-3} const?(typ′)
{-4} volatile?(typ(typ′))
{-5} non_bool_integral_enum?(typ(typ(typ′)))

{1} pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))

Expanding the definition of pod_data_type?,
Adding type constraints for dt_const(dt(typ(typ!1))),
we get 2 subgoals:

`dt_pod.3.1.1:`

{-1} interpreted_data_type?[(range(typ(typ′)))](dt_const(dt(typ(typ′))))
{-2} (∀ (*l*: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_1) ≡
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_2))
∧ size(uidt(dt_const(dt(typ(typ′))))) > 0
{-3} Cpp_Type?(typ′)
{-4} const?(typ′)
{-5} volatile?(typ(typ′))
{-6} non_bool_integral_enum?(typ(typ(typ′)))

{1} interpreted_data_type?(dt_const(dt(typ(typ′)))) ∧
(∀ (*l*: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_1) ≡
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_2))
∧ size(uidt(dt_const(dt(typ(typ′))))) > 0

Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_pod.3.1.1.1:`

{-1} uninterpreted_data_type?(dt_const(dt(typ(typ′)))`uidt)
{-2} ∀ (*d*: ((range(typ(typ′)))), *a*: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))
(to_byte(dt_const(dt(typ(typ′))))(*d*, *a*), *a*)
{-3} ∀ (*l*: list[Byte], *a*: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, *a*) ≡
up?(from_byte(dt_const(dt(typ(typ′))))(*l*, *a*))
{-4} ∀ (*d*: ((range(typ(typ′)))), *a*: Address):
down(from_byte(dt_const(dt(typ(typ′))))
(to_byte(dt_const(dt(typ(typ′))))(*d*, *a*), *a*))
= *d*
{-5} ∀ (*l*: list[Byte], a_1, a_2: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_1) ≡
valid?(uidt(dt_const(dt(typ(typ′)))))(*l*, a_2)
{-6} size(uidt(dt_const(dt(typ(typ′))))) > 0
{-7} Cpp_Type?(typ′)
{-8} const?(typ′)
{-9} volatile?(typ(typ′))
{-10} non_bool_integral_enum?(typ(typ(typ′)))

{1} ∀ (*d*: ((range(typ′))), *a*: Address):
valid?(uidt(dt_const(dt(typ(typ′)))))
(to_byte(dt_const(dt(typ(typ′))))(*d*, *a*), *a*)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_pod.3.1.1.1`.
`dt_pod.3.1.1.2:`

```
{-1}  uninterpreted_data_type?(dt_const(dt(typ(typ')))`uidt)
{-2}  ∀ (d: ((range(typ(typ')))), a: Address):
        valid?(uidt(dt_const(dt(typ(typ')))))
            (to_byte(dt_const(dt(typ(typ'))))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const(dt(typ(typ')))))(l, a) ≡
          up?(from_byte(dt_const(dt(typ(typ'))))(l, a))
{-4}  ∀ (d: ((range(typ(typ')))), a: Address):
        down(from_byte(dt_const(dt(typ(typ'))))
                    (to_byte(dt_const(dt(typ(typ'))))(d, a), a))
          = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_const(dt(typ(typ')))))(l, a1) ≡
          valid?(uidt(dt_const(dt(typ(typ')))))(l, a2)
{-6}  size(uidt(dt_const(dt(typ(typ'))))) > 0
{-7}  Cpp_Type?(typ')
{-8}  const?(typ')
{-9}  volatile?(typ(typ'))
{-10} non_bool_integral_enum?(typ(typ(typ')))
------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        down(from_byte(dt_const(dt(typ(typ'))))
                    (to_byte(dt_const(dt(typ(typ'))))(d, a), a))
          = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_pod.3.1.1.2`.
`dt_pod.3.1.2:`

```
{-1}  (∀ (l: list[Byte], a1, a2: Address):
          valid?(uidt(dt_const(dt(typ(typ')))))(l, a1) ≡
            valid?(uidt(dt_const(dt(typ(typ')))))(l, a2))
        ∧ size(uidt(dt_const(dt(typ(typ'))))) > 0
{-2}  Cpp_Type?(typ')
{-3}  const?(typ')
{-4}  volatile?(typ(typ'))
{-5}  non_bool_integral_enum?(typ(typ(typ')))
------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[(range(typ(typ')))])): TRUE
{2}   interpreted_data_type?(dt_const(dt(typ(typ')))) ∧
        (∀ (l: list[Byte], a1, a2: Address):
            valid?(uidt(dt_const(dt(typ(typ')))))(l, a1) ≡
              valid?(uidt(dt_const(dt(typ(typ')))))(l, a2))
          ∧ size(uidt(dt_const(dt(typ(typ'))))) > 0
```

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),
which is trivially true.

This completes the proof of `dt_pod.3.1.2`.

`dt_pod.3.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  const?(typ′)
{-3}  volatile?(typ(typ′))
{-4}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   pod_data_type?[(range(typ(typ′)))](dt(typ(typ′)))
{2}   pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))
```

Hiding formulas: (-2 2),

Expanding the definition of dt,

Using lemma dt_volatile_pod[(range(typ(typ(typ′))))],

we get 3 subgoals:

`dt_pod.3.2.1:`

```
{-1}  pod_data_type?[(range(typ(typ(typ′))))](dt_volatile(dt(typ(typ(typ′)))))
{-2}  Cpp_Type?(typ′)
{-3}  volatile?(typ(typ′))
{-4}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   pod_data_type?[(range(typ(typ′)))](dt_volatile(dt(typ(typ(typ′)))))
```

Expanding the definition of pod_data_type?,

Adding type constraints for dt_volatile(dt(typ(typ(typ!1)))),

we get 2 subgoals:

`dt_pod.3.2.1.1:`

```
{-1}  interpreted_data_type?[(range(typ(typ(typ′))))]
        (dt_volatile(dt(typ(typ(typ′)))))
{-2}  (∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a1) ≡
          valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a2))
      ∧ size(uidt(dt_volatile(dt(typ(typ(typ′))))))  > 0
{-3}  Cpp_Type?(typ′)
{-4}  volatile?(typ(typ′))
{-5}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   interpreted_data_type?(dt_volatile(dt(typ(typ(typ′)))))  ∧
        (∀ (l: list[Byte], a1, a2: Address):
          valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a1) ≡
            valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a2))
        ∧ size(uidt(dt_volatile(dt(typ(typ(typ′))))))  > 0
```

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_pod.3.2.1.1.1:`

```
{-1}  uninterpreted_data_type?(dt_volatile(dt(typ(typ(typ'))))`uidt)
{-2}  ∀ (d: ((range(typ(typ(typ'))))), a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))
            (to_byte(dt_volatile(dt(typ(typ(typ'))))))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a) ≡
          up?(from_byte(dt_volatile(dt(typ(typ(typ'))))))(l, a))
{-4}  ∀ (d: ((range(typ(typ(typ'))))), a: Address):
        down(from_byte(dt_volatile(dt(typ(typ(typ'))))))
                    (to_byte(dt_volatile(dt(typ(typ(typ'))))))(d, a), a))
          = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a1) ≡
          valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a2)
{-6}  size(uidt(dt_volatile(dt(typ(typ(typ')))))) > 0
{-7}  Cpp_Type?(typ')
{-8}  volatile?(typ(typ'))
{-9}  non_bool_integral_enum?(typ(typ(typ')))
-------------------------------------------------------------------------
{1}   ∀ (d: ((range(typ(typ'))))), a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))
            (to_byte(dt_volatile(dt(typ(typ(typ'))))))(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_pod.3.2.1.1.1`.
`dt_pod.3.2.1.1.2:`

```
{-1}  uninterpreted_data_type?(dt_volatile(dt(typ(typ(typ'))))`uidt)
{-2}  ∀ (d: ((range(typ(typ(typ'))))), a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))
            (to_byte(dt_volatile(dt(typ(typ(typ'))))))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a) ≡
          up?(from_byte(dt_volatile(dt(typ(typ(typ'))))))(l, a))
{-4}  ∀ (d: ((range(typ(typ(typ'))))), a: Address):
        down(from_byte(dt_volatile(dt(typ(typ(typ'))))))
                    (to_byte(dt_volatile(dt(typ(typ(typ'))))))(d, a), a))
          = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a1) ≡
          valid?(uidt(dt_volatile(dt(typ(typ(typ'))))))(l, a2)
{-6}  size(uidt(dt_volatile(dt(typ(typ(typ')))))) > 0
{-7}  Cpp_Type?(typ')
{-8}  volatile?(typ(typ'))
{-9}  non_bool_integral_enum?(typ(typ(typ')))
-------------------------------------------------------------------------
{1}   ∀ (d: ((range(typ(typ'))))), a: Address):
        down(from_byte(dt_volatile(dt(typ(typ(typ'))))))
                    (to_byte(dt_volatile(dt(typ(typ(typ'))))))(d, a), a))
          = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_pod.3.2.1.1.2`.
`dt_pod.3.2.1.2:`

```
{-1}  (∀ (l: list[Byte], a1, a2: Address):
         valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a1) ≡
          valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a2))
       ∧ size(uidt(dt_volatile(dt(typ(typ(typ′))))))  > 0
{-2}  Cpp_Type?(typ′)
{-3}  volatile?(typ(typ′))
{-4}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[(range(typ(typ(typ′))))])): TRUE
{2}   interpreted_data_type?(dt_volatile(dt(typ(typ(typ′))))) ∧
       (∀ (l: list[Byte], a1, a2: Address):
          valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a1) ≡
           valid?(uidt(dt_volatile(dt(typ(typ(typ′))))))(l, a2))
        ∧ size(uidt(dt_volatile(dt(typ(typ(typ′))))))  > 0
```

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ(typ!1)))),
which is trivially true.
This completes the proof of `dt_pod.3.2.1.2`.
`dt_pod.3.2.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  volatile?(typ(typ′))
{-3}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   pod_data_type?[(range(typ(typ(typ′))))](dt(typ(typ(typ′))))
{2}   pod_data_type?[(range(typ(typ′)))](dt_volatile(dt(typ(typ(typ′)))))
```

Hiding formulas: (-2 2),
Expanding the definition of dt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_pod.3.2.2.1:`

```
{-1}  const?(typ(typ(typ′)))
{-2}  Cpp_Type?(typ′)
{-3}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   pod_data_type?[(range(typ(typ(typ′))))](dt_const(dt(typ(typ(typ(typ′))))))
```

Keeping (-1 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.3.2.2.1`.
`dt_pod.3.2.2.2:`

```
{-1}  volatile?(typ(typ(typ′)))
{-2}  Cpp_Type?(typ′)
{-3}  non_bool_integral_enum?(typ(typ(typ′)))
------------------------------------------------------------------
{1}   pod_data_type?[(range(typ(typ(typ′))))]
        (dt_volatile(dt(typ(typ(typ(typ′))))))
```

Keeping (-1 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.3.2.2.2`.

`dt_pod.3.2.3:`

{-1}	Cpp_Type?(typ′)
{-2}	volatile?(typ(typ′))
{-3}	non_bool_integral_enum?(typ(typ(typ′)))
{1}	∃ (x_1 : (interpreted_data_type?[(range(typ(typ(typ′))))])): TRUE
{2}	pod_data_type?[(range(typ(typ′)))](dt_volatile(dt(typ(typ(typ′)))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ(typ!1)))),

which is trivially true.

This completes the proof of `dt_pod.3.2.3`.

`dt_pod.3.3:`

{-1}	Cpp_Type?(typ′)
{-2}	const?(typ′)
{-3}	volatile?(typ(typ′))
{-4}	non_bool_integral_enum?(typ(typ(typ′)))
{1}	∃ (x_1 : (interpreted_data_type?[(range(typ(typ′)))])): TRUE
{2}	pod_data_type?[((range(typ′)))](dt_const(dt(typ(typ′))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),

which is trivially true.

This completes the proof of `dt_pod.3.3`.

`dt_pod.4:`

{-1}	Cpp_Type?(typ′)
{-2}	non_bool_integral_enum?(typ(typ′))
{-3}	volatile?(typ′)
{1}	pod_data_type?[((range(typ′)))](dt(typ′))

Expanding the definition of dt,

Using lemma dt_volatile_pod[(range(typ(typ′)))],

we get 3 subgoals:

`dt_pod.4.1:`

{-1}	pod_data_type?[(range(typ(typ′)))](dt_volatile(dt(typ(typ′))))
{-2}	Cpp_Type?(typ′)
{-3}	non_bool_integral_enum?(typ(typ′))
{-4}	volatile?(typ′)
{1}	pod_data_type?[((range(typ′)))](dt_volatile(dt(typ(typ′))))

Expanding the definition of pod_data_type?,

Adding type constraints for dt_volatile(dt(typ(typ!1))),

we get 2 subgoals:

`dt_pod.4.1.1:`

```
{-1}  interpreted_data_type?[(range(typ(typ')))](dt_volatile(dt(typ(typ'))))
{-2}  (∀ (l: list[Byte], a1, a2: Address):
         valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a1) ≡
           valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a2))
        ∧ size(uidt(dt_volatile(dt(typ(typ'))))) > 0
{-3}  Cpp_Type?(typ')
{-4}  non_bool_integral_enum?(typ(typ'))
{-5}  volatile?(typ')
------------------------------------------------------------------------
{1}   interpreted_data_type?(dt_volatile(dt(typ(typ')))) ∧
        (∀ (l: list[Byte], a1, a2: Address):
           valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a1) ≡
             valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a2))
          ∧ size(uidt(dt_volatile(dt(typ(typ'))))) > 0
```

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_pod.4.1.1.1:`

```
{-1}  uninterpreted_data_type?(dt_volatile(dt(typ(typ')))`uidt)
{-2}  ∀ (d: ((range(typ(typ')))), a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))
             (to_byte(dt_volatile(dt(typ(typ'))))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a) ≡
          up?(from_byte(dt_volatile(dt(typ(typ'))))(l, a))
{-4}  ∀ (d: ((range(typ(typ')))), a: Address):
        down(from_byte(dt_volatile(dt(typ(typ'))))
                      (to_byte(dt_volatile(dt(typ(typ'))))(d, a), a))
          = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a1) ≡
          valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a2)
{-6}  size(uidt(dt_volatile(dt(typ(typ'))))) > 0
{-7}  Cpp_Type?(typ')
{-8}  non_bool_integral_enum?(typ(typ'))
{-9}  volatile?(typ')
------------------------------------------------------------------------
{1}   ∀ (d: ((range(typ(typ')))), a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))
             (to_byte(dt_volatile(dt(typ(typ'))))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of range,

which is trivially true.

This completes the proof of `dt_pod.4.1.1.1`.

`dt_pod.4.1.1.2:`

```
{-1}  uninterpreted_data_type?(dt_volatile(dt(typ(typ')))`uidt)
{-2}  ∀ (d: ((range(typ(typ')))), a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))
            (to_byte(dt_volatile(dt(typ(typ'))))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a) ≡
          up?(from_byte(dt_volatile(dt(typ(typ'))))(l, a))
{-4}  ∀ (d: ((range(typ(typ')))), a: Address):
        down(from_byte(dt_volatile(dt(typ(typ'))))
                    (to_byte(dt_volatile(dt(typ(typ'))))(d, a), a))
          = d
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a1) ≡
          valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a2)
{-6}  size(uidt(dt_volatile(dt(typ(typ'))))) > 0
{-7}  Cpp_Type?(typ')
{-8}  non_bool_integral_enum?(typ(typ'))
{-9}  volatile?(typ')
-------------------------------------------------------------------
{1}   ∀ (d: ((range(typ'))), a: Address):
        down(from_byte(dt_volatile(dt(typ(typ'))))
                    (to_byte(dt_volatile(dt(typ(typ'))))(d, a), a))
          = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of range,
which is trivially true.
This completes the proof of `dt_pod.4.1.1.2`.
`dt_pod.4.1.2:`

```
{-1}  (∀ (l: list[Byte], a1, a2: Address):
          valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a1) ≡
            valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a2))
        ∧ size(uidt(dt_volatile(dt(typ(typ'))))) > 0
{-2}  Cpp_Type?(typ')
{-3}  non_bool_integral_enum?(typ(typ'))
{-4}  volatile?(typ')
-------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[(range(typ(typ')))])): TRUE
{2}   interpreted_data_type?(dt_volatile(dt(typ(typ')))) ∧
        (∀ (l: list[Byte], a1, a2: Address):
            valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a1) ≡
              valid?(uidt(dt_volatile(dt(typ(typ')))))(l, a2))
          ∧ size(uidt(dt_volatile(dt(typ(typ'))))) > 0
```

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),
which is trivially true.
This completes the proof of `dt_pod.4.1.2`.
`dt_pod.4.2:`

```
{-1}  Cpp_Type?(typ')
{-2}  non_bool_integral_enum?(typ(typ'))
{-3}  volatile?(typ')
-------------------------------------------------------------------
{1}   pod_data_type?[(range(typ(typ')))](dt(typ(typ')))
{2}   pod_data_type?[((range(typ')))](dt_volatile(dt(typ(typ'))))
```

Hiding formulas: (-3 2),
Expanding the definition of dt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_pod.4.2.1:`

{-1} const?(typ(typ′))
{-2} Cpp_Type?(typ′)
{-3} non_bool_integral_enum?(typ(typ′))

{1} pod_data_type?[(range(typ(typ′)))](dt_const(dt(typ(typ(typ′)))))

Keeping (-1 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.4.2.1`.
`dt_pod.4.2.2:`

{-1} volatile?(typ(typ′))
{-2} Cpp_Type?(typ′)
{-3} non_bool_integral_enum?(typ(typ′))

{1} pod_data_type?[(range(typ(typ′)))](dt_volatile(dt(typ(typ(typ′)))))

Keeping (-1 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_pod.4.2.2`.
`dt_pod.4.3:`

{-1} Cpp_Type?(typ′)
{-2} non_bool_integral_enum?(typ(typ′))
{-3} volatile?(typ′)

{1} ∃ (x_1: (interpreted_data_type?[(range(typ(typ′)))])): TRUE
{2} pod_data_type?[((range(typ′)))](dt_volatile(dt(typ(typ′))))

Instantiating the top quantifier in 1 with the terms: (dt(typ(typ!1))),
which is trivially true.
This completes the proof of `dt_pod.4.3`.
Q.E.D.

C.44.39 Cpp_Types.dt_cv_pointer_TCC1

Terse proof for `dt_cv_pointer_TCC1`.
`dt_cv_pointer_TCC1:`

{1} well_founded?(λ (x: Cpp_Subtype(cv(pointer?)), y: Cpp_Subtype(cv(pointer?))):
restrict
[[Cpp_Type_, Cpp_Type_],
[Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))], boolean]
(≪)(x, y))

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(pointer?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `dt_cv_pointer_TCC1`.
Q.E.D.

C.44.40 Cpp_Types.dt_cv_pointer_TCC2

Terse proof for `dt_cv_pointer_TCC2`.

`dt_cv_pointer_TCC2:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) ⊃ Cpp_Type?(t) ∧ cv(pointer?)(t)

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_pointer_TCC2.1:`

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ′ = const(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC2.1`.

`dt_cv_pointer_TCC2.2:`

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ′ = const(t')
{1}	cv(pointer?)(t')

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC2.2`.
Q.E.D.

C.44.41 Cpp_Types.dt_cv_pointer_TCC3

Terse proof for `dt_cv_pointer_TCC3`.

`dt_cv_pointer_TCC3:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) ⊃ restrict [[Cpp_Type_, Cpp_Type_], [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))], boolean] (≪)(t, typ)

Repeatedly Skolemizing and flattening,
Expanding the definition of restrict,
Replacing using formula -3,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `dt_cv_pointer_TCC3`.

Q.E.D.

C.44.42 Cpp_Types.dt_cv_pointer_TCC4

Terse proof for `dt_cv_pointer_TCC4`.

`dt_cv_pointer_TCC4:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_pointer_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC4`.
Q.E.D.

C.44.43 Cpp_Types.dt_cv_pointer_TCC5

Terse proof for `dt_cv_pointer_TCC5`.

`dt_cv_pointer_TCC5:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) ⊃ pointer?(cv_base(t))

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Using lemma cv_base_result,
we get 2 subgoals:

`dt_cv_pointer_TCC5.1:`

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ$'$ = const(t')
{1}	cv(pointer?)(t')
{2}	pointer?(cv_base(t'))

Using lemma dt_cv_pointer_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC5.1`.

`dt_cv_pointer_TCC5.2:`

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ$'$ = const(t')
{1}	(pointer? ⊆ interpreted?)
{2}	pointer?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC5.2`.
Q.E.D.

C.44.44 Cpp_Types.dt_cv_pointer_TCC6

Terse proof for `dt_cv_pointer_TCC6`.

`dt_cv_pointer_TCC6:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = const(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range_pointer(cv_base(t))))])): TRUE)

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(t!1))),

we get 3 subgoals:

`dt_cv_pointer_TCC6.1:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer?)(typ′)
{-3}	typ′ = const(t')
{1}	TRUE

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC6.1`.

`dt_cv_pointer_TCC6.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer?)(typ′)
{-3}	typ′ = const(t')
{1}	pointer?(cv_base(t'))

Using lemma dt_cv_pointer_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC6.2`.

`dt_cv_pointer_TCC6.3:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer?)(typ′)
{-3}	typ′ = const(t')
{1}	Cpp_Type?(t')

Using lemma dt_cv_pointer_TCC4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC6.3`.

Q.E.D.

C.44.45 Cpp_Types.dt_cv_pointer_TCC7

Terse proof for `dt_cv_pointer_TCC7`.

dt_cv_pointer_TCC7:

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)),
          v:
            [typ1:
               {z: Cpp_Subtype(cv(pointer?)) |
                     restrict
                       [[Cpp_Type_, Cpp_Type_],
                        [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))],
                        boolean]
                       (≪)(z, typ)} →
                 (interpreted_data_type?[((range_pointer(cv_base(typ1))))])],
          t: Cpp_Type_):
        typ = const(t) ⊃
         (∀ (x: lift[Pointer_Base_Type]):
            range_pointer(cv_base(t))(x) ≡ range_pointer(cv_base(typ))(x))
          ∧
          (∀ (x: lift[Pointer_Base_Type]):
             range_pointer(cv_base(t))(x) ≡ range_pointer(cv_base(typ))(x))
           ∧
           (∀ (x1: [list[Byte], Address]):
              every[((range_pointer(cv_base(t))))]
                   ((range_pointer(cv_base(typ))))
                   (dt_const[((range_pointer(cv_base(t))))](v(t))`from_byte(x1)))
            ∧
            interpreted_data_type?[(range_pointer(cv_base(typ)))]
                 (dt_const[((range_pointer(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of cv_base,

Applying propositional simplification,

we get 2 subgoals:

dt_cv_pointer_TCC7.1:

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(pointer?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------
{1}  ∀ (x1: [list[Byte], Address]):
       every[((range_pointer(cv_base(t'))))]
            ((range_pointer(cv_base(const(t')))))
            (dt_const[((range_pointer(cv_base(t'))))](v'(t'))`from_byte(x1))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of dt_cv_pointer_TCC7.1.

`dt_cv_pointer_TCC7.2:`

{-1}	Cpp_Type?(const(t'))
{-2}	cv(pointer?)(const(t'))
{-3}	typ$'$ = const(t')
{1}	interpreted_data_type?[((range_pointer(cv_base(typ$'$))))] (dt_const[((range_pointer(cv_base(t'))))](v'(t')))

Adding type constraints for dt_const(v!1(t!1)),

we get 3 subgoals:

`dt_cv_pointer_TCC7.2.1:`

{-1}	interpreted_data_type?[((range_pointer(cv_base(t'))))](dt_const(v'(t')))
{-2}	Cpp_Type?(const(t'))
{-3}	cv(pointer?)(const(t'))
{-4}	typ$'$ = const(t')
{1}	interpreted_data_type?[((range_pointer(cv_base(typ$'$))))] (dt_const[((range_pointer(cv_base(t'))))](v'(t')))

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_cv_pointer_TCC7.2.1.1:`

{-1}	uninterpreted_data_type?(dt_const(v'(t'))`uidt)
{-2}	∀ (d: ((range_pointer(cv_base(t')))), a: Address): valid?(uidt(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}	∀ (l: list[Byte], a: Address): valid?(uidt(dt_const(v'(t')))(l, a) ≡ up?(from_byte(dt_const(v'(t')))(l, a))
{-4}	∀ (d: ((range_pointer(cv_base(t')))), a: Address): down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)) = d
{-5}	Cpp_Type?(const(t'))
{-6}	cv(pointer?)(const(t'))
{-7}	typ$'$ = const(t')
{1}	∀ (d: ((range_pointer(cv_base(typ$'$)))), a: Address): valid?(uidt(dt_const[((range_pointer(cv_base(t'))))](v'(t')))) (to_byte(dt_const[((range_pointer(cv_base(t'))))](v'(t')))(d, a), a)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC7.2.1.1`.

`dt_cv_pointer_TCC7.2.1.2:`

```
{-1}  uninterpreted_data_type?(dt_const(v'(t'))`uidt)
{-2}  ∀ (d: ((range_pointer(cv_base(t')))), a: Address):
        valid?(uidt(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const(v'(t')))(l, a) ≡
         up?(from_byte(dt_const(v'(t')))(l, a))
{-4}  ∀ (d: ((range_pointer(cv_base(t')))), a: Address):
        down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)) =
         d
{-5}  Cpp_Type?(const(t'))
{-6}  cv(pointer?)(const(t'))
{-7}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_pointer(cv_base(typ')))), a: Address):
        down(from_byte(dt_const[((range_pointer(cv_base(t'))))](v'(t')))
                    (to_byte(dt_const[((range_pointer(cv_base(t'))))](v'(t')))
                        (d, a),
                      a))
         = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC7.2.1.2`.

`dt_cv_pointer_TCC7.2.2:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(pointer?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[((range_pointer(cv_base(t'))))])): TRUE
{2}   interpreted_data_type?[((range_pointer(cv_base(typ'))))]
          (dt_const[((range_pointer(cv_base(t'))))](v'(t')))
```

Using lemma dt_cv_pointer_TCC6,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC7.2.2`.

`dt_cv_pointer_TCC7.2.3:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(pointer?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   pointer?(cv_base(t'))
{2}   interpreted_data_type?[((range_pointer(cv_base(typ'))))]
          (dt_const[((range_pointer(cv_base(t'))))](v'(t')))
```

Using lemma dt_cv_pointer_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC7.2.3`.

Q.E.D.

C.44.46 Cpp_Types.dt_cv_pointer_TCC8

Terse proof for `dt_cv_pointer_TCC8`.

`dt_cv_pointer_TCC8:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_):
       typ = volatile(t) ⊃ Cpp_Type?(t) ∧ cv(pointer?)(t)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_pointer_TCC8.1:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(pointer?)(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------
{1}   Cpp_Type?(t')
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC8.1`.

`dt_cv_pointer_TCC8.2:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(pointer?)(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------
{1}   cv(pointer?)(t')
```

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC8.2`.
Q.E.D.

C.44.47 Cpp_Types.dt_cv_pointer_TCC9

Terse proof for `dt_cv_pointer_TCC9`.

`dt_cv_pointer_TCC9:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_):
       typ = volatile(t) ⊃
        restrict
            [[Cpp_Type_, Cpp_Type_],
              [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))], boolean]
            (≪)(t, typ)
```

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of ■,
which is trivially true.

This completes the proof of `dt_cv_pointer_TCC9`.
Q.E.D.

C.44.48 Cpp_Types.dt_cv_pointer_TCC10

Terse proof for `dt_cv_pointer_TCC10`.
`dt_cv_pointer_TCC10:`

{1} ∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_): typ = volatile(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_pointer_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC10`.
Q.E.D.

C.44.49 Cpp_Types.dt_cv_pointer_TCC11

Terse proof for `dt_cv_pointer_TCC11`.
`dt_cv_pointer_TCC11:`

{1} ∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_):
typ = volatile(t) ⊃ pointer?(cv_base(t))

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_pointer_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC11`.
Q.E.D.

C.44.50 Cpp_Types.dt_cv_pointer_TCC12

Terse proof for `dt_cv_pointer_TCC12`.
`dt_cv_pointer_TCC12:`

{1} ∀ (typ: Cpp_Subtype(cv(pointer?)), t: Cpp_Type_):
typ = volatile(t) ⊃
(∃ (x_1: (interpreted_data_type?[((range_pointer(cv_base(t))))])): TRUE)

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (dt_pointer(cv_base(t!1))),
we get 3 subgoals:
`dt_cv_pointer_TCC12.1:`

{-1} Cpp_Type?(typ′)
{-2} cv(pointer?)(typ′)
{-3} typ′ = volatile(t')
{1} TRUE

which is trivially true.

This completes the proof of `dt_cv_pointer_TCC12.1`.

`dt_cv_pointer_TCC12.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer?)(typ′)
{-3}	typ′ = volatile(t')
{1}	pointer?(cv_base(t'))

Using lemma dt_cv_pointer_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC12.2`.

`dt_cv_pointer_TCC12.3:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer?)(typ′)
{-3}	typ′ = volatile(t')
{1}	Cpp_Type?(t')

Using lemma dt_cv_pointer_TCC8,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC12.3`.

Q.E.D.

C.44.51 Cpp_Types.dt_cv_pointer_TCC13

Terse proof for `dt_cv_pointer_TCC13`.

dt_cv_pointer_TCC13:

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer?)),
          v:
            [typ1:
                {z: Cpp_Subtype(cv(pointer?)) |
                            restrict
                                 [[Cpp_Type_, Cpp_Type_],
                                   [Cpp_Subtype(cv(pointer?)), Cpp_Subtype(cv(pointer?))],
                                   boolean]
                                 (≪)(z, typ)} →
                   (interpreted_data_type?[((range_pointer(cv_base(typ1))))])],
          t: Cpp_Type_):
        typ = volatile(t) ⊃
         (∀ (x: lift[Pointer_Base_Type]):
             range_pointer(cv_base(t))(x) ≡ range_pointer(cv_base(typ))(x))
          ∧
          (∀ (x: lift[Pointer_Base_Type]):
              range_pointer(cv_base(t))(x) ≡ range_pointer(cv_base(typ))(x))
           ∧
           (∀ (x1: [list[Byte], Address]):
               every[((range_pointer(cv_base(t))))]
                    ((range_pointer(cv_base(typ))))
                    (dt_volatile[((range_pointer(cv_base(t))))](v(t))`from_byte
                          (x1)))
            ∧
            interpreted_data_type?[((range_pointer(cv_base(typ))))]
                 (dt_volatile[((range_pointer(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of cv_base,
Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

dt_cv_pointer_TCC13.1:

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(pointer?)(volatile(t'))
{-3}  typ' = volatile(t')
-------------------------------------------------------------------
{1}   interpreted_data_type?[((range_pointer(cv_base(typ'))))]
          (dt_volatile[((range_pointer(cv_base(t'))))](v'(t')))
```

Adding type constraints for dt_volatile(v!1(t!1)),
we get 3 subgoals:

dt_cv_pointer_TCC13.1.1:

```
{-1}  interpreted_data_type?[((range_pointer(cv_base(t'))))](dt_volatile(v'(t')))
{-2}  Cpp_Type?(volatile(t'))
{-3}  cv(pointer?)(volatile(t'))
{-4}  typ' = volatile(t')
-------------------------------------------------------------------
{1}   interpreted_data_type?[((range_pointer(cv_base(typ'))))]
          (dt_volatile[((range_pointer(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_cv_pointer_TCC13.1.1.1:`

{-1} uninterpreted_data_type?(dt_volatile($v'(t')$)`uidt)
{-2} ∀ (d: ((range_pointer(cv_base(t')))), a: Address):
valid?(uidt(dt_volatile($v'(t')$)))(to_byte(dt_volatile($v'(t')$))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_volatile($v'(t')$)))(l, a) ≡
up?(from_byte(dt_volatile($v'(t')$))(l, a))
{-4} ∀ (d: ((range_pointer(cv_base(t')))), a: Address):
down(from_byte(dt_volatile($v'(t')$))
(to_byte(dt_volatile($v'(t')$))(d, a), a))
= d
{-5} Cpp_Type?(volatile(t'))
{-6} cv(pointer?)(volatile(t'))
{-7} typ′ = volatile(t')

{1} ∀ (d: ((range_pointer(cv_base(typ′)))), a: Address):
valid?(uidt(dt_volatile[((range_pointer(cv_base(t'))))]($v'(t')$))
(to_byte(dt_volatile[((range_pointer(cv_base(t'))))]($v'(t')$))(d, a),
a)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_pointer_TCC13.1.1.1`.
`dt_cv_pointer_TCC13.1.1.2:`

{-1} uninterpreted_data_type?(dt_volatile($v'(t')$)`uidt)
{-2} ∀ (d: ((range_pointer(cv_base(t')))), a: Address):
valid?(uidt(dt_volatile($v'(t')$)))(to_byte(dt_volatile($v'(t')$))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_volatile($v'(t')$)))(l, a) ≡
up?(from_byte(dt_volatile($v'(t')$))(l, a))
{-4} ∀ (d: ((range_pointer(cv_base(t')))), a: Address):
down(from_byte(dt_volatile($v'(t')$))
(to_byte(dt_volatile($v'(t')$))(d, a), a))
= d
{-5} Cpp_Type?(volatile(t'))
{-6} cv(pointer?)(volatile(t'))
{-7} typ′ = volatile(t')

{1} ∀ (d: ((range_pointer(cv_base(typ′)))), a: Address):
down(from_byte(dt_volatile[((range_pointer(cv_base(t'))))]($v'(t')$))
(to_byte(dt_volatile[((range_pointer(cv_base(t'))))]($v'(t')$))
(d, a),
a))
= d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,

which is trivially true.
This completes the proof of `dt_cv_pointer_TCC13.1.1.2`.
`dt_cv_pointer_TCC13.1.2`:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer?)(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range_pointer(cv_base(t'))))])): TRUE
{2}	interpreted_data_type?[((range_pointer(cv_base(typ$'$))))] (dt_volatile[((range_pointer(cv_base(t'))))](v'(t')))

Using lemma dt_cv_pointer_TCC12,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC13.1.2`.
`dt_cv_pointer_TCC13.1.3`:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer?)(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	pointer?(cv_base(t'))
{2}	interpreted_data_type?[((range_pointer(cv_base(typ$'$))))] (dt_volatile[((range_pointer(cv_base(t'))))](v'(t')))

Using lemma dt_cv_pointer_TCC11,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC13.1.3`.
`dt_cv_pointer_TCC13.2`:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer?)(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\forall$ (x_1: [list[Byte], Address]): every[((range_pointer(cv_base(t'))))] ((range_pointer(cv_base(volatile(t'))))) (dt_volatile[((range_pointer(cv_base(t'))))](v'(t'))`from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_pointer_TCC13.2`.
Q.E.D.

C.44.52 Cpp_Types.dt_cv_pointer_TCC14

Terse proof for `dt_cv_pointer_TCC14`.
`dt_cv_pointer_TCC14`:

{1}	$\forall$ (typ: Cpp_Subtype(cv(pointer?))): $\neg$ volatile?(typ) $\wedge$ $\neg$ const?(typ) $\supset$ pointer?(typ)

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_cv_pointer_TCC14`.
Q.E.D.

C.44.53 Cpp_Types.dt_cv_pointer_TCC15

Terse proof for `dt_cv_pointer_TCC15`.
`dt_cv_pointer_TCC15:`

$$
\begin{array}{ll}
\{1\} & \forall\ (\mathrm{typ}\colon\ \mathrm{Cpp\_Subtype}(\mathrm{cv}(\mathrm{pointer?}))):\\
& \quad \neg\ \mathrm{volatile?}(\mathrm{typ})\ \wedge\ \neg\ \mathrm{const?}(\mathrm{typ})\ \supset\\
& \quad\ (\forall\ (x\colon\ \mathrm{lift}[\mathrm{Pointer\_Base\_Type}])\colon\\
& \qquad \mathrm{range\_pointer}(\mathrm{typ})(x)\ \equiv\ \mathrm{range\_pointer}(\mathrm{cv\_base}(\mathrm{typ}))(x))\\
& \quad\ \wedge\\
& \quad\ (\forall\ (x\colon\ \mathrm{lift}[\mathrm{Pointer\_Base\_Type}])\colon\\
& \qquad \mathrm{range\_pointer}(\mathrm{typ})(x)\ \equiv\ \mathrm{range\_pointer}(\mathrm{cv\_base}(\mathrm{typ}))(x))\\
& \quad\ \wedge\\
& \quad\ (\forall\ (x_1\colon\ [\mathrm{list}[\mathrm{Byte}],\ \mathrm{Address}])\colon\\
& \qquad \mathrm{every}[((\mathrm{range\_pointer}(\mathrm{typ})))]\\
& \qquad\quad ((\mathrm{range\_pointer}(\mathrm{cv\_base}(\mathrm{typ}))))(\mathrm{dt\_pointer}(\mathrm{typ})\text{`}\mathrm{from\_byte}(x_1)))\\
& \quad\ \wedge\ \mathrm{interpreted\_data\_type?}[(\mathrm{range\_pointer}(\mathrm{cv\_base}(\mathrm{typ})))](\mathrm{dt\_pointer}(\mathrm{typ}))
\end{array}
$$

Repeatedly Skolemizing and flattening,
Using lemma cv_base_result2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_cv_pointer_TCC15.1:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{cv\_base}(\mathrm{typ}')\ =\ \mathrm{typ}'\\
\{-2\} & \mathrm{Cpp\_Type?}(\mathrm{typ}')\\
\{-3\} & \mathrm{cv}(\mathrm{pointer?})(\mathrm{typ}')\\
\hline
\{1\} & \mathrm{const?}(\mathrm{typ}')\\
\{2\} & \mathrm{volatile?}(\mathrm{typ}')\\
\{3\} & \mathrm{interpreted\_data\_type?}[((\mathrm{range\_pointer}(\mathrm{cv\_base}(\mathrm{typ}'))))](\mathrm{dt\_pointer}(\mathrm{typ}'))
\end{array}
$$

Replacing using formula -1,
Adding type constraints for dt_pointer(cv_base(typ!1)),
Rewriting using pod_is_interpreted_data, matching in *,
This completes the proof of `dt_cv_pointer_TCC15.1`.
`dt_cv_pointer_TCC15.2:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{cv\_base}(\mathrm{typ}')\ =\ \mathrm{typ}'\\
\{-2\} & \mathrm{Cpp\_Type?}(\mathrm{typ}')\\
\{-3\} & \mathrm{cv}(\mathrm{pointer?})(\mathrm{typ}')\\
\hline
\{1\} & \mathrm{const?}(\mathrm{typ}')\\
\{2\} & \mathrm{volatile?}(\mathrm{typ}')\\
\{3\} & \forall\ (x_1\colon\ [\mathrm{list}[\mathrm{Byte}],\ \mathrm{Address}])\colon\\
& \quad \mathrm{every}[((\mathrm{range\_pointer}(\mathrm{typ}')))]\\
& \qquad ((\mathrm{range\_pointer}(\mathrm{cv\_base}(\mathrm{typ}'))))(\mathrm{dt\_pointer}(\mathrm{typ}')\text{`}\mathrm{from\_byte}(x_1))
\end{array}
$$

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_pointer_TCC15.2`.

dt_cv_pointer_TCC15.3:

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(typ′)
{-3}  cv(pointer?)(typ′)
-------------------------------------------------------------------------
{1}   const?(typ′)
{2}   volatile?(typ′)
{3}   ∀ (x: lift[Pointer_Base_Type]):
        range_pointer(typ′)(x) ≡ range_pointer(cv_base(typ′))(x)
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_pointer_TCC15.3.
Q.E.D.

C.44.54 Cpp_Types.dt_cv_bitfield_TCC1

Terse proof for dt_cv_bitfield_TCC1.

dt_cv_bitfield_TCC1:

```
-------------------------------------------------------------------------
{1}   well_founded?(λ (x: Cpp_Subtype(cv(bitfield?)), y: Cpp_Subtype(cv(bitfield?))):
                      restrict
                        [[Cpp_Type_, Cpp_Type_],
                         [Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))], boolean]
                        (≪)(x, y))
```

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(bitfield?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of dt_cv_bitfield_TCC1.
Q.E.D.

C.44.55 Cpp_Types.dt_cv_bitfield_TCC2

Terse proof for dt_cv_bitfield_TCC2.

dt_cv_bitfield_TCC2:

```
-------------------------------------------------------------------------
{1}   ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_):
        typ = const(t) ⊃ Cpp_Type?(t) ∧ cv(bitfield?)(t)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Applying propositional simplification,
we get 2 subgoals:

dt_cv_bitfield_TCC2.1:

```
{-1}  Cpp_Type?(const(t′))
{-2}  cv(bitfield?)(const(t′))
{-3}  typ′ = const(t′)
-------------------------------------------------------------------------
{1}   Cpp_Type?(t′)
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),

Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC2.1`.
`dt_cv_bitfield_TCC2.2`:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(bitfield?)(const(t'))
{-3}	typ$'$ = const(t')
{1}	cv(bitfield?)(t')

Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC2.2`.
Q.E.D.

C.44.56 Cpp_Types.dt_cv_bitfield_TCC3

Terse proof for `dt_cv_bitfield_TCC3`.
`dt_cv_bitfield_TCC3`:

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_):
typ = const(t) ⊃
restrict
[[Cpp_Type_, Cpp_Type_],
[Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))], boolean]
(≪)(t, typ)

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `dt_cv_bitfield_TCC3`.
Q.E.D.

C.44.57 Cpp_Types.dt_cv_bitfield_TCC4

Terse proof for `dt_cv_bitfield_TCC4`.
`dt_cv_bitfield_TCC4`:

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_): typ = const(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_bitfield_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC4`.
Q.E.D.

C.44.58 Cpp_Types.dt_cv_bitfield_TCC5

Terse proof for `dt_cv_bitfield_TCC5`.

`dt_cv_bitfield_TCC5:`

|
|---|
| {1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_): typ = const(t) ⊃ bitfield?(cv_base(t)) |

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:
`dt_cv_bitfield_TCC5.1:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(bitfield?)(typ′)
{-3}	typ′ = const(t')
{1}	cv(bitfield?)(t')
{2}	bitfield?(cv_base(t'))

Using lemma dt_cv_bitfield_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC5.1`.
`dt_cv_bitfield_TCC5.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(bitfield?)(typ′)
{-3}	typ′ = const(t')
{1}	(bitfield? ⊆ interpreted?)
{2}	bitfield?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC5.2`.
Q.E.D.

C.44.59 Cpp_Types.dt_cv_bitfield_TCC6

Terse proof for `dt_cv_bitfield_TCC6`.
`dt_cv_bitfield_TCC6:`

|
|---|
| {1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_): typ = const(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range_bitfield(cv_base(t))))])): TRUE) |

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_bitfield_TCC2,
Using lemma dt_bitfield_exists,
we get 2 subgoals:
`dt_cv_bitfield_TCC6.1:`

{-1}	typ′ = const(t') ⊃ Cpp_Type?(t') ∧ cv(bitfield?)(t')
{-2}	Cpp_Type?(typ′)
{-3}	cv(bitfield?)(typ′)
{-4}	typ′ = const(t')
{1}	bitfield?(cv_base(t'))
{2}	∃ (x_1: (interpreted_data_type?[((range_bitfield(cv_base(t'))))])): TRUE

Using lemma dt_cv_bitfield_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC6.1`.
`dt_cv_bitfield_TCC6.2`:

```
{-1}  typ' = const(t') ⊃ Cpp_Type?(t') ∧ cv(bitfield?)(t')
{-2}  Cpp_Type?(typ')
{-3}  cv(bitfield?)(typ')
{-4}  typ' = const(t')
------------------------------------------------------------
{1}   Cpp_Type?(t')
{2}   ∃ (x1: (interpreted_data_type?[((range_bitfield(cv_base(t'))))])): TRUE
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC6.2`.
Q.E.D.

C.44.60 Cpp_Types.dt_cv_bitfield_TCC7

Terse proof for `dt_cv_bitfield_TCC7`.
`dt_cv_bitfield_TCC7`:

```
{1}  ∀ (typ: Cpp_Subtype(cv(bitfield?)),
          v:
            [typ1:
                {z: Cpp_Subtype(cv(bitfield?)) |
                            restrict
                                [[Cpp_Type_, Cpp_Type_],
                                  [Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))],
                                  boolean]
                                (≪)(z, typ)} →
                    (interpreted_data_type?[((range_bitfield(cv_base(typ1))))])],
          t: Cpp_Type_):
       typ = const(t) ⊃
        (∀ (x: int):
             range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x))
         ∧
         (∀ (x: int):
              range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x))
          ∧
          (∀ (x1: [list[Byte], Address]):
               every[((range_bitfield(cv_base(t))))]
                    ((range_bitfield(cv_base(typ))))
                    (dt_const[((range_bitfield(cv_base(t))))](v(t))‘from_byte(x1)))
           ∧
           interpreted_data_type?[(range_bitfield(cv_base(typ)))]
                (dt_const[((range_bitfield(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of cv_base,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_bitfield_TCC7.1:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(bitfield?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (x1: [list[Byte], Address]):
        every[((range_bitfield(cv_base(t'))))]
             ((range_bitfield(cv_base(const(t')))))
             (dt_const[((range_bitfield(cv_base(t'))))](v'(t'))`from_byte(x1))
```

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_bitfield_TCC7.1`.
`dt_cv_bitfield_TCC7.2:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(bitfield?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
          (dt_const[((range_bitfield(cv_base(t'))))](v'(t')))
```

Adding type constraints for dt_const(v!1(t!1)),
we get 3 subgoals:
`dt_cv_bitfield_TCC7.2.1:`

```
{-1}  interpreted_data_type?[((range_bitfield(cv_base(t'))))](dt_const(v'(t')))
{-2}  Cpp_Type?(const(t'))
{-3}  cv(bitfield?)(const(t'))
{-4}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
          (dt_const[((range_bitfield(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:
`dt_cv_bitfield_TCC7.2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_const(v'(t'))`uidt)
{-2}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        valid?(uidt(dt_const(v'(t'))))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const(v'(t'))))(l, a) ≡
          up?(from_byte(dt_const(v'(t')))(l, a))
{-4}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)) =
          d
{-5}  Cpp_Type?(const(t'))
{-6}  cv(bitfield?)(const(t'))
{-7}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_bitfield(cv_base(typ')))), a: Address):
        valid?(uidt(dt_const[((range_bitfield(cv_base(t'))))](v'(t'))))
             (to_byte(dt_const[((range_bitfield(cv_base(t'))))](v'(t')))(d, a),
               a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_bitfield_TCC7.2.1.1`.
`dt_cv_bitfield_TCC7.2.1.2:`

```
{-1}  uninterpreted_data_type?(dt_const(v'(t'))`uidt)
{-2}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        valid?(uidt(dt_const(v'(t'))))(to_byte(dt_const(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const(v'(t'))))(l, a) ≡
          up?(from_byte(dt_const(v'(t')))(l, a))
{-4}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        down(from_byte(dt_const(v'(t')))(to_byte(dt_const(v'(t')))(d, a), a)) =
          d
{-5}  Cpp_Type?(const(t'))
{-6}  cv(bitfield?)(const(t'))
{-7}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_bitfield(cv_base(typ')))), a: Address):
        down(from_byte(dt_const[((range_bitfield(cv_base(t'))))](v'(t')))
                      (to_byte(dt_const[((range_bitfield(cv_base(t'))))](v'(t')))
                               (d, a),
                             a))
          = d
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_bitfield_TCC7.2.1.2`.
`dt_cv_bitfield_TCC7.2.2:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(bitfield?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[((range_bitfield(cv_base(t'))))])): TRUE
{2}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
          (dt_const[((range_bitfield(cv_base(t'))))](v'(t')))
```

Using lemma dt_cv_bitfield_TCC6,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC7.2.2`.
`dt_cv_bitfield_TCC7.2.3:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(bitfield?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   bitfield?(cv_base(t'))
{2}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
          (dt_const[((range_bitfield(cv_base(t'))))](v'(t')))
```

Using lemma dt_cv_bitfield_TCC5,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of dt_cv_bitfield_TCC7.2.3.
Q.E.D.

C.44.61 Cpp_Types.dt_cv_bitfield_TCC8

Terse proof for dt_cv_bitfield_TCC8.
dt_cv_bitfield_TCC8:

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_):
typ = volatile(t) ⊃ Cpp_Type?(t) ∧ cv(bitfield?)(t)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
dt_cv_bitfield_TCC8.1:

{-1} Cpp_Type?(typ′)
{-2} cv(bitfield?)(typ′)
{-3} typ′ = volatile(t')
{1} Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Replacing using formula -4,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_bitfield_TCC8.1.
dt_cv_bitfield_TCC8.2:

{-1} Cpp_Type?(typ′)
{-2} cv(bitfield?)(typ′)
{-3} typ′ = volatile(t')
{1} cv(bitfield?)(t')

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_bitfield_TCC8.2.
Q.E.D.

C.44.62 Cpp_Types.dt_cv_bitfield_TCC9

Terse proof for dt_cv_bitfield_TCC9.
dt_cv_bitfield_TCC9:

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_):
typ = volatile(t) ⊃
restrict
[[Cpp_Type_, Cpp_Type_],
[Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))], boolean]
(≪)(t, typ)

Repeatedly Skolemizing and flattening,
Replacing using formula -3,

Expanding the definition of restrict,
Expanding the definition of ■,
which is trivially true.
This completes the proof of `dt_cv_bitfield_TCC9`.
Q.E.D.

C.44.63 Cpp_Types.dt_cv_bitfield_TCC10

Terse proof for `dt_cv_bitfield_TCC10`.
`dt_cv_bitfield_TCC10:`

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_): typ = volatile(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_bitfield_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC10`.
Q.E.D.

C.44.64 Cpp_Types.dt_cv_bitfield_TCC11

Terse proof for `dt_cv_bitfield_TCC11`.
`dt_cv_bitfield_TCC11:`

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_):
typ = volatile(t) ⊃ bitfield?(cv_base(t))

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_bitfield_TCC8,
Using lemma cv_base_result,
we get 2 subgoals:
`dt_cv_bitfield_TCC11.1:`

{-1} typ′ = volatile(t') ⊃ Cpp_Type?(t') ∧ cv(bitfield?)(t')
{-2} Cpp_Type?(typ′)
{-3} cv(bitfield?)(typ′)
{-4} typ′ = volatile(t')

{1} cv(bitfield?)(t')
{2} bitfield?(cv_base(t'))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC11.1`.
`dt_cv_bitfield_TCC11.2:`

{-1} typ′ = volatile(t') ⊃ Cpp_Type?(t') ∧ cv(bitfield?)(t')
{-2} Cpp_Type?(typ′)
{-3} cv(bitfield?)(typ′)
{-4} typ′ = volatile(t')

{1} (bitfield? ⊆ interpreted?)
{2} bitfield?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC11.2`.

Q.E.D.

C.44.65 Cpp_Types.dt_cv_bitfield_TCC12

Terse proof for `dt_cv_bitfield_TCC12`.

`dt_cv_bitfield_TCC12:`

{1} ∀ (typ: Cpp_Subtype(cv(bitfield?)), t: Cpp_Type_):
typ = volatile(t) ⊃
($\exists$ (x_1: (interpreted_data_type?[((range_bitfield(cv_base(t))))])): TRUE)

Repeatedly Skolemizing and flattening,

Using lemma dt_bitfield_exists,

we get 2 subgoals:

`dt_cv_bitfield_TCC12.1:`

{-1} Cpp_Type?(typ′)
{-2} cv(bitfield?)(typ′)
{-3} typ′ = volatile(t')

{1} bitfield?(cv_base(t'))
{2} $\exists$ (x_1: (interpreted_data_type?[((range_bitfield(cv_base(t'))))])): TRUE

Using lemma dt_cv_bitfield_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_bitfield_TCC12.1`.

`dt_cv_bitfield_TCC12.2:`

{-1} Cpp_Type?(typ′)
{-2} cv(bitfield?)(typ′)
{-3} typ′ = volatile(t')

{1} Cpp_Type?(t')
{2} $\exists$ (x_1: (interpreted_data_type?[((range_bitfield(cv_base(t'))))])): TRUE

Using lemma dt_cv_bitfield_TCC8,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_bitfield_TCC12.2`.

Q.E.D.

C.44.66 Cpp_Types.dt_cv_bitfield_TCC13

Terse proof for `dt_cv_bitfield_TCC13`.

`dt_cv_bitfield_TCC13:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(bitfield?)),
          v:
            [typ1:
               {z: Cpp_Subtype(cv(bitfield?)) |
                     restrict
                       [[Cpp_Type_, Cpp_Type_],
                        [Cpp_Subtype(cv(bitfield?)), Cpp_Subtype(cv(bitfield?))],
                        boolean]
                       (≪)(z, typ)} →
                 (interpreted_data_type?[((range_bitfield(cv_base(typ1))))])],
          t: Cpp_Type_):
        typ = volatile(t) ⊃
         (∀ (x: int):
            range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x))
          ∧
          (∀ (x: int):
             range_bitfield(cv_base(t))(x) ≡ range_bitfield(cv_base(typ))(x))
           ∧
           (∀ (x1: [list[Byte], Address]):
              every[((range_bitfield(cv_base(t))))]
                   ((range_bitfield(cv_base(typ))))
                   (dt_volatile[((range_bitfield(cv_base(t))))](v(t))`from_byte
                         (x1)))
            ∧
            interpreted_data_type?[(range_bitfield(cv_base(typ)))]
              (dt_volatile[((range_bitfield(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Expanding the definition of cv_base,

Applying propositional simplification,

we get 2 subgoals:

`dt_cv_bitfield_TCC13.1:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(bitfield?)(volatile(t'))
{-3}  typ' = volatile(t')
-------------------------------------------------------------------------
{1}  ∀ (x1: [list[Byte], Address]):
       every[((range_bitfield(cv_base(t'))))]
            ((range_bitfield(cv_base(volatile(t')))))
            (dt_volatile[((range_bitfield(cv_base(t'))))](v'(t'))`from_byte(x1))
```

Expanding the definition of every,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_bitfield_TCC13.1`.

`dt_cv_bitfield_TCC13.2:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(bitfield?)(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------------
{1}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
        (dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))
```

Adding type constraints for dt_volatile(v!1(t!1)),

we get 3 subgoals:

`dt_cv_bitfield_TCC13.2.1:`

```
{-1}  interpreted_data_type?[((range_bitfield(cv_base(t'))))](dt_volatile(v'(t')))
{-2}  Cpp_Type?(volatile(t'))
{-3}  cv(bitfield?)(volatile(t'))
{-4}  typ' = volatile(t')
------------------------------------------------------------------
{1}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
        (dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_cv_bitfield_TCC13.2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_volatile(v'(t'))`uidt)
{-2}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(to_byte(dt_volatile(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(l, a) ≡
          up?(from_byte(dt_volatile(v'(t')))(l, a))
{-4}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        down(from_byte(dt_volatile(v'(t')))
                      (to_byte(dt_volatile(v'(t')))(d, a), a))
          = d
{-5}  Cpp_Type?(volatile(t'))
{-6}  cv(bitfield?)(volatile(t'))
{-7}  typ' = volatile(t')
------------------------------------------------------------------
{1}   ∀ (d: ((range_bitfield(cv_base(typ')))), a: Address):
        valid?(uidt(dt_volatile[((range_bitfield(cv_base(t'))))](v'(t'))))
          (to_byte(dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))(d, a),
           a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_bitfield_TCC13.2.1.1`.

`dt_cv_bitfield_TCC13.2.1.2:`

```
{-1}  uninterpreted_data_type?(dt_volatile(v'(t'))`uidt)
{-2}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(to_byte(dt_volatile(v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile(v'(t'))))(l, a) ≡
          up?(from_byte(dt_volatile(v'(t')))(l, a))
{-4}  ∀ (d: ((range_bitfield(cv_base(t')))), a: Address):
        down(from_byte(dt_volatile(v'(t')))
                      (to_byte(dt_volatile(v'(t')))(d, a), a))
          = d
{-5}  Cpp_Type?(volatile(t'))
{-6}  cv(bitfield?)(volatile(t'))
{-7}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_bitfield(cv_base(typ')))), a: Address):
        down(from_byte(dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))
                      (to_byte(dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))
                            (d, a),
                       a))
          = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_bitfield_TCC13.2.1.2`.

`dt_cv_bitfield_TCC13.2.2:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(bitfield?)(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[((range_bitfield(cv_base(t'))))])): TRUE
{2}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
          (dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))
```

Using lemma dt_cv_bitfield_TCC12,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_bitfield_TCC13.2.2`.

`dt_cv_bitfield_TCC13.2.3:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(bitfield?)(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   bitfield?(cv_base(t'))
{2}   interpreted_data_type?[((range_bitfield(cv_base(typ'))))]
          (dt_volatile[((range_bitfield(cv_base(t'))))](v'(t')))
```

Using lemma dt_cv_bitfield_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_bitfield_TCC13.2.3`.

Q.E.D.

C.44.67 Cpp_Types.dt_cv_bitfield_TCC14

Terse proof for `dt_cv_bitfield_TCC14`.

`dt_cv_bitfield_TCC14:`

$$\frac{}{\begin{array}{ll}\{1\} & \forall\ (\text{typ}\colon\ \text{Cpp\_Subtype(cv(bitfield?))})\colon \\ & \neg\ \text{volatile?(typ)} \wedge \neg\ \text{const?(typ)} \supset \text{bitfield?(typ)}\end{array}}$$

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `dt_cv_bitfield_TCC14`.

Q.E.D.

C.44.68 Cpp_Types.dt_cv_bitfield_TCC15

Terse proof for `dt_cv_bitfield_TCC15`.

`dt_cv_bitfield_TCC15:`

$$\frac{}{\begin{array}{ll}\{1\} & \forall\ (\text{typ}\colon\ \text{Cpp\_Subtype(cv(bitfield?))})\colon \\ & \neg\ \text{volatile?(typ)} \wedge \neg\ \text{const?(typ)} \supset \\ & (\forall\ (x\colon\ \text{int})\colon\ \text{range\_bitfield(typ)}(x) \equiv \text{range\_bitfield(cv\_base(typ))}(x)) \wedge \\ & (\forall\ (x\colon\ \text{int})\colon\ \text{range\_bitfield(typ)}(x) \equiv \text{range\_bitfield(cv\_base(typ))}(x)) \wedge \\ & (\forall\ (x_1\colon\ [\text{list}[\text{Byte}],\ \text{Address}])\colon \\ & \quad \text{every}[((\text{range\_bitfield(typ)}))] \\ & \quad\quad ((\text{range\_bitfield(cv\_base(typ))}))(\text{dt\_bitfield(typ)`from\_byte}(x_1))) \\ & \wedge\ \text{interpreted\_data\_type?}[(\text{range\_bitfield(cv\_base(typ))})](\text{dt\_bitfield(typ)})\end{array}}$$

Repeatedly Skolemizing and flattening,

Using lemma cv_base_result2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`dt_cv_bitfield_TCC15.1:`

$$\frac{\begin{array}{ll}\{-1\} & \text{cv\_base}(\text{typ}') = \text{typ}' \\ \{-2\} & \text{Cpp\_Type?}(\text{typ}') \\ \{-3\} & \text{cv(bitfield?)}(\text{typ}')\end{array}}{\begin{array}{ll}\{1\} & \text{const?}(\text{typ}') \\ \{2\} & \text{volatile?}(\text{typ}') \\ \{3\} & \text{interpreted\_data\_type?}[((\text{range\_bitfield(cv\_base}(\text{typ}'))))](\text{dt\_bitfield}(\text{typ}'))\end{array}}$$

Replacing using formula -1,

Adding type constraints for dt_bitfield(cv_base(typ!1)),

which is trivially true.

This completes the proof of `dt_cv_bitfield_TCC15.1`.

`dt_cv_bitfield_TCC15.2`:

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(typ′)
{-3}  cv(bitfield?)(typ′)
------------------------------------------------------------------------
{1}   const?(typ′)
{2}   volatile?(typ′)
{3}   ∀ (x1: [list[Byte], Address]):
        every[((range_bitfield(typ′)))]
          ((range_bitfield(cv_base(typ′))))(dt_bitfield(typ′)‘from_byte(x1))
```

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC15.2`.
`dt_cv_bitfield_TCC15.3`:

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(typ′)
{-3}  cv(bitfield?)(typ′)
------------------------------------------------------------------------
{1}   const?(typ′)
{2}   volatile?(typ′)
{3}   ∀ (x: int): range_bitfield(typ′)(x) ≡ range_bitfield(cv_base(typ′))(x)
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_bitfield_TCC15.3`.
Q.E.D.

C.44.69 Cpp_Types.dt_cv_ptm_TCC1

Terse proof for `dt_cv_ptm_TCC1`.
`dt_cv_ptm_TCC1`:

```
------------------------------------------------------------------------
{1}   well_founded?(λ (x: Cpp_Subtype(cv(pointer_to_member?)),
                         y: Cpp_Subtype(cv(pointer_to_member?))):
                      restrict
                        [[Cpp_Type_, Cpp_Type_],
                          [Cpp_Subtype(cv(pointer_to_member?)),
                            Cpp_Subtype(cv(pointer_to_member?))],
                          boolean]
                        (≪)(x, y))
```

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(pointer_to_member?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `dt_cv_ptm_TCC1`.
Q.E.D.

C.44.70 Cpp_Types.dt_cv_ptm_TCC2

Terse proof for `dt_cv_ptm_TCC2`.

dt_cv_ptm_TCC2:

{1} ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
typ = const(t) ⊃ Cpp_Type?(t) ∧ cv(pointer_to_member?)(t)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
dt_cv_ptm_TCC2.1:

{-1} Cpp_Type?(typ′)
{-2} cv(pointer_to_member?)(typ′)
{-3} typ′ = const(t′)
{1} Cpp_Type?(t′)

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Replacing using formula -4,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_ptm_TCC2.1.
dt_cv_ptm_TCC2.2:

{-1} Cpp_Type?(typ′)
{-2} cv(pointer_to_member?)(typ′)
{-3} typ′ = const(t′)
{1} cv(pointer_to_member?)(t′)

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_ptm_TCC2.2.
Q.E.D.

C.44.71 Cpp_Types.dt_cv_ptm_TCC3

Terse proof for dt_cv_ptm_TCC3.
dt_cv_ptm_TCC3:

{1} ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
typ = const(t) ⊃
restrict
[[Cpp_Type_, Cpp_Type_],
[Cpp_Subtype(cv(pointer_to_member?)), Cpp_Subtype(cv(pointer_to_member?))],
boolean]
(≪)(t, typ)

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of ■,
which is trivially true.
This completes the proof of dt_cv_ptm_TCC3.
Q.E.D.

C.44.72 Cpp_Types.dt_cv_ptm_TCC4

Terse proof for `dt_cv_ptm_TCC4`.

`dt_cv_ptm_TCC4:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_): typ = const(t) ⊃ Cpp_Type?(t)

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_ptm_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC4`.
Q.E.D.

C.44.73 Cpp_Types.dt_cv_ptm_TCC5

Terse proof for `dt_cv_ptm_TCC5`.

`dt_cv_ptm_TCC5:`

{1}	∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_): typ = const(t) ⊃ pointer_to_member?(cv_base(t))

Repeatedly Skolemizing and flattening,
Using lemma cv_base_result,
we get 2 subgoals:

`dt_cv_ptm_TCC5.1:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer_to_member?)(typ′)
{-3}	typ′ = const(t')
{1}	cv(pointer_to_member?)(t')
{2}	pointer_to_member?(cv_base(t'))

Using lemma dt_cv_ptm_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC5.1`.

`dt_cv_ptm_TCC5.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(pointer_to_member?)(typ′)
{-3}	typ′ = const(t')
{1}	(pointer_to_member? ⊆ interpreted?)
{2}	pointer_to_member?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC5.2`.
Q.E.D.

C.44.74 Cpp_Types.dt_cv_ptm_TCC6

Terse proof for `dt_cv_ptm_TCC6`.

`dt_cv_ptm_TCC6:`

|
|---|
| {1} ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_): typ = const(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range_ptm(cv_base(t))))])): TRUE) |

Repeatedly Skolemizing and flattening,

Using lemma dt_pointer_to_member_exists,

we get 3 subgoals:

`dt_cv_ptm_TCC6.1:`

{-1}	∃ (dt: (pod_data_type?[((range_ptm(cv_base(t'))))])): TRUE
{-2}	Cpp_Type?(typ$'$)
{-3}	cv(pointer_to_member?)(typ$'$)
{-4}	typ$'$ = const(t')
{1}	∃ (x_1: (interpreted_data_type?[((range_ptm(cv_base(t'))))])): TRUE

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `dt_cv_ptm_TCC6.1`.

`dt_cv_ptm_TCC6.2:`

{-1}	Cpp_Type?(typ$'$)
{-2}	cv(pointer_to_member?)(typ$'$)
{-3}	typ$'$ = const(t')
{1}	pointer_to_member?(cv_base(t'))
{2}	∃ (x_1: (interpreted_data_type?[((range_ptm(cv_base(t'))))])): TRUE

Using lemma dt_cv_ptm_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC6.2`.

`dt_cv_ptm_TCC6.3:`

{-1}	Cpp_Type?(typ$'$)
{-2}	cv(pointer_to_member?)(typ$'$)
{-3}	typ$'$ = const(t')
{1}	Cpp_Type?(t')
{2}	∃ (x_1: (interpreted_data_type?[((range_ptm(cv_base(t'))))])): TRUE

Using lemma dt_cv_ptm_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC6.3`.

Q.E.D.

C.44.75 Cpp_Types.dt_cv_ptm_TCC7

Terse proof for `dt_cv_ptm_TCC7`.

`dt_cv_ptm_TCC7:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)),
          v:
            [typ1:
                {z: Cpp_Subtype(cv(pointer_to_member?)) |
                        restrict
                            [[Cpp_Type_, Cpp_Type_],
                              [Cpp_Subtype(cv(pointer_to_member?)),
                                Cpp_Subtype(cv(pointer_to_member?))],
                              boolean]
                            (≪)(z, typ)} →
                   (interpreted_data_type?[((range_ptm(cv_base(typ1))))])],
            t: Cpp_Type_):
       typ = const(t) ⊃
        (∀ (x: nat): range_ptm(cv_base(t))(x) ≡ range_ptm(cv_base(typ))(x)) ∧
         (∀ (x: nat): range_ptm(cv_base(t))(x) ≡ range_ptm(cv_base(typ))(x)) ∧
          (∀ (x_1: [list[Byte], Address]):
              every[((range_ptm(cv_base(t))))]
                   ((range_ptm(cv_base(typ))))
                   (dt_const[((range_ptm(cv_base(t))))](v(t))`from_byte(x_1)))
           ∧
           interpreted_data_type?[(range_ptm(cv_base(typ)))]
               (dt_const[((range_ptm(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of cv_base,
Applying propositional simplification,
we get 2 subgoals:
`dt_cv_ptm_TCC7.1:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(pointer_to_member?)(const(t'))
{-3}  typ' = const(t')
-------------------------------------------------------------------------
{1}   ∀ (x_1: [list[Byte], Address]):
        every[((range_ptm(cv_base(t'))))]
             ((range_ptm(cv_base(const(t')))))
             (dt_const[((range_ptm(cv_base(t'))))](v'(t'))`from_byte(x_1))
```

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_ptm_TCC7.1`.
`dt_cv_ptm_TCC7.2:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(pointer_to_member?)(const(t'))
{-3}  typ' = const(t')
-------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_ptm(cv_base(typ'))))]
          (dt_const[((range_ptm(cv_base(t'))))](v'(t')))
```

Adding type constraints for dt_const[((range_ptm(cv_base(t!1))))](v!1(t!1)),

we get 3 subgoals:

`dt_cv_ptm_TCC7.2.1:`

```
{-1}  interpreted_data_type?[((range_ptm(cv_base(t')))) ]
        (dt_const[((range_ptm(cv_base(t'))))](v'(t')))
{-2}  Cpp_Type?(const(t'))
{-3}  cv(pointer_to_member?)(const(t'))
{-4}  typ' = const(t')
------------------------------------------------------------
{1}   interpreted_data_type?[((range_ptm(cv_base(typ')))) ]
        (dt_const[((range_ptm(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,

Applying propositional simplification,

we get 2 subgoals:

`dt_cv_ptm_TCC7.2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_const[((range_ptm(cv_base(t'))))](v'(t'))`uidt)
{-2}  ∀ (d: (((range_ptm(cv_base(t'))))), a: Address):
        valid?(uidt(dt_const[((range_ptm(cv_base(t'))))](v'(t'))))
          (to_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const[((range_ptm(cv_base(t'))))](v'(t'))))(l, a) ≡
         up?(from_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(l, a))
{-4}  ∀ (d: (((range_ptm(cv_base(t'))))), a: Address):
        down(from_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))
                 (to_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(d, a),
                   a))
          = d
{-5}  Cpp_Type?(const(t'))
{-6}  cv(pointer_to_member?)(const(t'))
{-7}  typ' = const(t')
------------------------------------------------------------
{1}   ∀ (d: ((range_ptm(cv_base(typ')))), a: Address):
        valid?(uidt(dt_const[((range_ptm(cv_base(t'))))](v'(t'))))
          (to_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(d, a), a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -7,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_ptm_TCC7.2.1.1`.

`dt_cv_ptm_TCC7.2.1.2:`

{-1} uninterpreted_data_type?(dt_const[((range_ptm(cv_base(t'))))](v'(t'))`uidt)
{-2} ∀ (d: (((range_ptm(cv_base(t'))))), a: Address):
valid?(uidt(dt_const[((range_ptm(cv_base(t'))))](v'(t'))))
(to_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(d, a), a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_const[((range_ptm(cv_base(t'))))](v'(t'))))(l, a) ≡
up?(from_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(l, a))
{-4} ∀ (d: (((range_ptm(cv_base(t'))))), a: Address):
down(from_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))
(to_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(d, a),
a))
= d
{-5} Cpp_Type?(const(t'))
{-6} cv(pointer_to_member?)(const(t'))
{-7} typ′ = const(t')

{1} ∀ (d: ((range_ptm(cv_base(typ′)))), a: Address):
down(from_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))
(to_byte(dt_const[((range_ptm(cv_base(t'))))](v'(t')))(d, a),
a))
= d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_ptm_TCC7.2.1.2`.
`dt_cv_ptm_TCC7.2.2:`

{-1} Cpp_Type?(const(t'))
{-2} cv(pointer_to_member?)(const(t'))
{-3} typ′ = const(t')

{1} ∃ (x_1: (interpreted_data_type?[((range_ptm(cv_base(t'))))])): TRUE
{2} interpreted_data_type?[((range_ptm(cv_base(typ′))))]
(dt_const[((range_ptm(cv_base(t'))))](v'(t')))

Using lemma dt_cv_ptm_TCC6,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC7.2.2`.
`dt_cv_ptm_TCC7.2.3:`

{-1} Cpp_Type?(const(t'))
{-2} cv(pointer_to_member?)(const(t'))
{-3} typ′ = const(t')

{1} pointer_to_member?(cv_base(t'))
{2} interpreted_data_type?[((range_ptm(cv_base(typ′))))]
(dt_const[((range_ptm(cv_base(t'))))](v'(t')))

Using lemma dt_cv_ptm_TCC5,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC7.2.3`.
Q.E.D.

C.44.76 Cpp_Types.dt_cv_ptm_TCC8

Terse proof for `dt_cv_ptm_TCC8`.

`dt_cv_ptm_TCC8:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
       typ = volatile(t) ⊃ Cpp_Type?(t) ∧ cv(pointer_to_member?)(t)
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_ptm_TCC8.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(pointer_to_member?)(typ′)
{-3}  typ′ = volatile(t′)
{1}   Cpp_Type?(t′)
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Replacing using formula -4,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC8.1`.

`dt_cv_ptm_TCC8.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(pointer_to_member?)(typ′)
{-3}  typ′ = volatile(t′)
{1}   cv(pointer_to_member?)(t′)
```

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC8.2`.
Q.E.D.

C.44.77 Cpp_Types.dt_cv_ptm_TCC9

Terse proof for `dt_cv_ptm_TCC9`.

`dt_cv_ptm_TCC9:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
       typ = volatile(t) ⊃
        restrict
            [[Cpp_Type_, Cpp_Type_],
              [Cpp_Subtype(cv(pointer_to_member?)), Cpp_Subtype(cv(pointer_to_member?))],
              boolean]
            (≪)(t, typ)
```

Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of ■,

which is trivially true.
This completes the proof of `dt_cv_ptm_TCC9`.
Q.E.D.

C.44.78 Cpp_Types.dt_cv_ptm_TCC10

Terse proof for `dt_cv_ptm_TCC10`.
`dt_cv_ptm_TCC10:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
       typ = volatile(t) ⊃ Cpp_Type?(t)
```

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_ptm_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC10`.
Q.E.D.

C.44.79 Cpp_Types.dt_cv_ptm_TCC11

Terse proof for `dt_cv_ptm_TCC11`.
`dt_cv_ptm_TCC11:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
       typ = volatile(t) ⊃ pointer_to_member?(cv_base(t))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:
`dt_cv_ptm_TCC11.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(pointer_to_member?)(typ′)
{-3}  typ′ = volatile(t′)
-------------------------------------
{1}   cv(pointer_to_member?)(t′)
{2}   pointer_to_member?(cv_base(t′))
```

Using lemma dt_cv_ptm_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC11.1`.
`dt_cv_ptm_TCC11.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(pointer_to_member?)(typ′)
{-3}  typ′ = volatile(t′)
-------------------------------------
{1}   (pointer_to_member? ⊆ interpreted?)
{2}   pointer_to_member?(cv_base(t′))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC11.2`.
Q.E.D.

C.44.80 Cpp_Types.dt_cv_ptm_TCC12

Terse proof for `dt_cv_ptm_TCC12`.

`dt_cv_ptm_TCC12:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)), t: Cpp_Type_):
       typ = volatile(t) ⊃
         (∃ (x1: (interpreted_data_type?[((range_ptm(cv_base(t))))])): TRUE)
```

Repeatedly Skolemizing and flattening,

Using lemma dt_pointer_to_member_exists,

we get 3 subgoals:

`dt_cv_ptm_TCC12.1:`

```
{-1}  ∃ (dt: (pod_data_type?[((range_ptm(cv_base(t′))))])): TRUE
{-2}  Cpp_Type?(typ′)
{-3}  cv(pointer_to_member?)(typ′)
{-4}  typ′ = volatile(t′)
------------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[((range_ptm(cv_base(t′))))])): TRUE
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `dt_cv_ptm_TCC12.1`.

`dt_cv_ptm_TCC12.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(pointer_to_member?)(typ′)
{-3}  typ′ = volatile(t′)
------------------------------------------------------------------------
{1}   pointer_to_member?(cv_base(t′))
{2}   ∃ (x1: (interpreted_data_type?[((range_ptm(cv_base(t′))))])): TRUE
```

Using lemma dt_cv_ptm_TCC11,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC12.2`.

`dt_cv_ptm_TCC12.3:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(pointer_to_member?)(typ′)
{-3}  typ′ = volatile(t′)
------------------------------------------------------------------------
{1}   Cpp_Type?(t′)
{2}   ∃ (x1: (interpreted_data_type?[((range_ptm(cv_base(t′))))])): TRUE
```

Using lemma dt_cv_ptm_TCC8,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_ptm_TCC12.3`.

Q.E.D.

C.44.81 Cpp_Types.dt_cv_ptm_TCC13

Terse proof for `dt_cv_ptm_TCC13`.

`dt_cv_ptm_TCC13:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?)),
          v:
            [typ1:
                {z: Cpp_Subtype(cv(pointer_to_member?)) |
                        restrict
                            [[Cpp_Type_, Cpp_Type_],
                              [Cpp_Subtype(cv(pointer_to_member?)),
                                Cpp_Subtype(cv(pointer_to_member?))],
                              boolean]
                            (≪)(z, typ)} →
                   (interpreted_data_type?[((range_ptm(cv_base(typ1))))])],
            t: Cpp_Type_):
       typ = volatile(t) ⊃
        (∀ (x: nat): range_ptm(cv_base(t))(x) ≡ range_ptm(cv_base(typ))(x)) ∧
         (∀ (x: nat): range_ptm(cv_base(t))(x) ≡ range_ptm(cv_base(typ))(x)) ∧
          (∀ (x1: [list[Byte], Address]):
              every[((range_ptm(cv_base(t))))]
                  ((range_ptm(cv_base(typ))))
                  (dt_volatile[((range_ptm(cv_base(t))))](v(t))`from_byte(x1)))
           ∧
           interpreted_data_type?[(range_ptm(cv_base(typ)))]
               (dt_volatile[((range_ptm(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of cv_base,
Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`dt_cv_ptm_TCC13.1:`

```
{-1}  Cpp_Type?(volatile(t'))
{-2}  cv(pointer_to_member?)(volatile(t'))
{-3}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_ptm(cv_base(typ'))))]
          (dt_volatile[((range_ptm(cv_base(t'))))](v'(t')))
```

Adding type constraints for dt_volatile[((range_ptm(cv_base(t!1))))](v!1(t!1)),
we get 3 subgoals:
`dt_cv_ptm_TCC13.1.1:`

```
{-1}  interpreted_data_type?[((range_ptm(cv_base(t'))))]
          (dt_volatile[((range_ptm(cv_base(t'))))](v'(t')))
{-2}  Cpp_Type?(volatile(t'))
{-3}  cv(pointer_to_member?)(volatile(t'))
{-4}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_ptm(cv_base(typ'))))]
          (dt_volatile[((range_ptm(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_ptm_TCC13.1.1.1:`

```
{-1}  uninterpreted_data_type?(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))`uidt)
{-2}  ∀ (d: (((range_ptm(cv_base(t')))))), a: Address):
        valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))))
            (to_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))))(l, a) ≡
          up?(from_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))(l, a))
{-4}  ∀ (d: (((range_ptm(cv_base(t')))))), a: Address):
        down(from_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))
                    (to_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))
                        (d, a),
                      a))
          = d
{-5}  Cpp_Type?(volatile(t'))
{-6}  cv(pointer_to_member?)(volatile(t'))
{-7}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_ptm(cv_base(typ')))), a: Address):
        valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))))
            (to_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))(d, a), a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_ptm_TCC13.1.1.1`.
`dt_cv_ptm_TCC13.1.1.2:`

```
{-1}  uninterpreted_data_type?(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))`uidt)
{-2}  ∀ (d: (((range_ptm(cv_base(t')))))), a: Address):
        valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))))
            (to_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile[((range_ptm(cv_base(t')))))](v'(t'))))(l, a) ≡
          up?(from_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))(l, a))
{-4}  ∀ (d: (((range_ptm(cv_base(t')))))), a: Address):
        down(from_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))
                    (to_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))
                        (d, a),
                      a))
          = d
{-5}  Cpp_Type?(volatile(t'))
{-6}  cv(pointer_to_member?)(volatile(t'))
{-7}  typ' = volatile(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_ptm(cv_base(typ')))), a: Address):
        down(from_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))
                    (to_byte(dt_volatile[((range_ptm(cv_base(t')))))](v'(t')))
                        (d, a),
                      a))
          = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,
Replacing using formula -7,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_ptm_TCC13.1.1.2`.
`dt_cv_ptm_TCC13.1.2`:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer_to_member?)(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\exists$ (x_1: (interpreted_data_type?[((range_ptm(cv_base(t'))))])): TRUE
{2}	interpreted_data_type?[((range_ptm(cv_base(typ$'$))))] (dt_volatile[((range_ptm(cv_base(t'))))](v'(t')))

Using lemma dt_cv_ptm_TCC12,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC13.1.2`.
`dt_cv_ptm_TCC13.1.3`:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer_to_member?)(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	pointer_to_member?(cv_base(t'))
{2}	interpreted_data_type?[((range_ptm(cv_base(typ$'$))))] (dt_volatile[((range_ptm(cv_base(t'))))](v'(t')))

Using lemma dt_cv_ptm_TCC11,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC13.1.3`.
`dt_cv_ptm_TCC13.2`:

{-1}	Cpp_Type?(volatile(t'))
{-2}	cv(pointer_to_member?)(volatile(t'))
{-3}	typ$'$ = volatile(t')
{1}	$\forall$ (x_1: [list[Byte], Address]): every[((range_ptm(cv_base(t'))))] ((range_ptm(cv_base(volatile(t'))))) (dt_volatile[((range_ptm(cv_base(t'))))](v'(t'))`from_byte(x_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_ptm_TCC13.2`.
Q.E.D.

C.44.82 Cpp_Types.dt_cv_ptm_TCC14

Terse proof for `dt_cv_ptm_TCC14`.
`dt_cv_ptm_TCC14`:

{1}	$\forall$ (typ: Cpp_Subtype(cv(pointer_to_member?))): $\neg$ volatile?(typ) $\wedge$ $\neg$ const?(typ) $\supset$ pointer_to_member?(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC14`.
Q.E.D.

C.44.83 Cpp_Types.dt_cv_ptm_TCC15

Terse proof for `dt_cv_ptm_TCC15`.
`dt_cv_ptm_TCC15:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?))):
       ¬ volatile?(typ) ∧ ¬ const?(typ) ⊃
        (∀ (x: nat): range_ptm(typ)(x) ≡ range_ptm(cv_base(typ))(x)) ∧
         (∀ (x: nat): range_ptm(typ)(x) ≡ range_ptm(cv_base(typ))(x)) ∧
          (∀ (x1: [list[Byte], Address]):
               every[((range_ptm(typ)))]
                    ((range_ptm(cv_base(typ))))(dt_ptm(typ)‘from_byte(x1)))
            ∧ interpreted_data_type?[(range_ptm(cv_base(typ)))](dt_ptm(typ))
```

Repeatedly Skolemizing and flattening,
Using lemma cv_base_result2,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_cv_ptm_TCC15.1:`

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(cv_base(typ′))
{-3}  cv(pointer_to_member?)(cv_base(typ′))
-------------------------------------------
{1}   volatile?(cv_base(typ′))
{2}   const?(cv_base(typ′))
{3}   interpreted_data_type?[((range_ptm(cv_base(typ′))))](dt_ptm(cv_base(typ′)))
```

Adding type constraints for dt_ptm(cv_base(typ!1)),
Rewriting using pod_is_interpreted_data, matching in *,
This completes the proof of `dt_cv_ptm_TCC15.1`.
`dt_cv_ptm_TCC15.2:`

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(cv_base(typ′))
{-3}  cv(pointer_to_member?)(cv_base(typ′))
-------------------------------------------
{1}   volatile?(cv_base(typ′))
{2}   const?(cv_base(typ′))
{3}   ∀ (x1: [list[Byte], Address]):
        every[((range_ptm(typ′)))]
             ((range_ptm(cv_base(cv_base(typ′)))))(dt_ptm(cv_base(typ′))‘from_byte(x1))
```

Expanding the definition of every,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_ptm_TCC15.2`.

dt_cv_ptm_TCC15.3:

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(cv_base(typ′))
{-3}  cv(pointer_to_member?)(cv_base(typ′))
------------------------------------------------------------
{1}   volatile?(cv_base(typ′))
{2}   const?(cv_base(typ′))
{3}   ∀ (x: nat):
        range_ptm(cv_base(typ′))(x) ≡ range_ptm(cv_base(cv_base(typ′)))(x)
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_ptm_TCC15.3.
Q.E.D.

C.44.84 Cpp_Types.dt_cv_float_TCC1

Terse proof for dt_cv_float_TCC1.
dt_cv_float_TCC1:

```
------------------------------------------------------------
{1}   well_founded?(λ (x: Cpp_Subtype(cv(floating_point?)),
                       y: Cpp_Subtype(cv(floating_point?))):
                      restrict
                        [[Cpp_Type_, Cpp_Type_],
                         [Cpp_Subtype(cv(floating_point?)),
                          Cpp_Subtype(cv(floating_point?))],
                         boolean]
                        (≪)(x, y))
```

Using lemma well_founded_restrict[Cpp_Type_, Cpp_Subtype(cv(floating_point?))],
Expanding the definition of restrict,
which is trivially true.
This completes the proof of dt_cv_float_TCC1.
Q.E.D.

C.44.85 Cpp_Types.dt_cv_float_TCC2

Terse proof for dt_cv_float_TCC2.
dt_cv_float_TCC2:

```
------------------------------------------------------------
{1}   ∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_):
        typ = const(t) ⊃ Cpp_Type?(t) ∧ cv(floating_point?)(t)
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
dt_cv_float_TCC2.1:

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(floating_point?)(typ′)
{-3}  typ′ = const(t′)
------------------------------------------------------------
{1}   Cpp_Type?(t′)
```

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Replacing using formula -4,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC2.1`.
`dt_cv_float_TCC2.2`:

```
{-1}  Cpp_Type?(typ')
{-2}  cv(floating_point?)(typ')
{-3}  typ' = const(t')
-------------------------------
{1}   cv(floating_point?)(t')
```

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_float_TCC2.2`.
Q.E.D.

C.44.86 Cpp_Types.dt_cv_float_TCC3

Terse proof for `dt_cv_float_TCC3`.
`dt_cv_float_TCC3`:

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_):
       typ = const(t) ⊃
        restrict
            [[Cpp_Type_, Cpp_Type_],
              [Cpp_Subtype(cv(floating_point?)), Cpp_Subtype(cv(floating_point?))], boolean]
            (≪)(t, typ)
```

Expanding the definition of ≪,
Expanding the definition of restrict,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC3`.
Q.E.D.

C.44.87 Cpp_Types.dt_cv_float_TCC4

Terse proof for `dt_cv_float_TCC4`.
`dt_cv_float_TCC4`:

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_):
       typ = const(t) ⊃ Cpp_Type?(t)
```

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_float_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC4`.
Q.E.D.

C.44.88 Cpp_Types.dt_cv_float_TCC5

Terse proof for `dt_cv_float_TCC5`.

`dt_cv_float_TCC5:`

{1}	∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = const(t) ⊃ floating_point?(cv_base(t))

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:

`dt_cv_float_TCC5.1:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = const(t')
{1}	cv(floating_point?)(t')
{2}	floating_point?(cv_base(t'))

Using lemma dt_cv_float_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC5.1`.

`dt_cv_float_TCC5.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = const(t')
{1}	(floating_point? ⊆ interpreted?)
{2}	floating_point?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_float_TCC5.2`.
Q.E.D.

C.44.89 Cpp_Types.dt_cv_float_TCC6

Terse proof for `dt_cv_float_TCC6`.

`dt_cv_float_TCC6:`

{1}	∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = const(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range_floating_point(cv_base(t))))])): TRUE)

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(t!1))),
we get 3 subgoals:

`dt_cv_float_TCC6.1:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = const(t')
{1}	TRUE

which is trivially true.

This completes the proof of `dt_cv_float_TCC6.1`.

`dt_cv_float_TCC6.2`:

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = const(t')
{1}	floating_point?(cv_base(t'))

Using lemma dt_cv_float_TCC5,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC6.2`.

`dt_cv_float_TCC6.3`:

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = const(t')
{1}	Cpp_Type?(t')

Using lemma dt_cv_float_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC6.3`.

Q.E.D.

C.44.90 Cpp_Types.dt_cv_float_TCC7

Terse proof for `dt_cv_float_TCC7`.

`dt_cv_float_TCC7:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)),
          v:
            [typ1:
                {z: Cpp_Subtype(cv(floating_point?)) |
                            restrict
                                [[Cpp_Type_, Cpp_Type_],
                                  [Cpp_Subtype(cv(floating_point?)),
                                    Cpp_Subtype(cv(floating_point?))],
                                  boolean]
                                (≪)(z, typ)} →
                   (interpreted_data_type?[((range_floating_point(cv_base(typ1))))])],
          t: Cpp_Type_):
        typ = const(t) ⊃
         (∀ (x: extended_real):
             range_floating_point(cv_base(t))(x) ≡
              range_floating_point(cv_base(typ))(x))
          ∧
          (∀ (x: extended_real):
             range_floating_point(cv_base(t))(x) ≡
              range_floating_point(cv_base(typ))(x))
          ∧
          (∀ (x₁: [list[Byte], Address]):
             every[((range_floating_point(cv_base(t))))]
                  ((range_floating_point(cv_base(typ))))
                  (dt_const[((range_floating_point(cv_base(t))))](v(t))`from_byte
                        (x₁)))
           ∧
           interpreted_data_type?[(range_floating_point(cv_base(typ)))]
              (dt_const[((range_floating_point(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Expanding the definition of cv_base,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_float_TCC7.1:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(floating_point?)(const(t'))
{-3}  typ' = const(t')
-------------------------------------------------------------------------
{1}  ∀ (x₁: [list[Byte], Address]):
       every[((range_floating_point(cv_base(t'))))]
            ((range_floating_point(cv_base(const(t')))))
            (dt_const[((range_floating_point(cv_base(t'))))](v'(t'))`from_byte(x₁))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_float_TCC7.1`.

`dt_cv_float_TCC7.2:`

```
{-1}  Cpp_Type?(const(t'))
{-2}  cv(floating_point?)(const(t'))
{-3}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_floating_point(cv_base(typ'))))]
        (dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
```

Adding type constraints for dt_const[((range_floating_point(cv_base(t!1))))](v!1(t!1)),
we get 3 subgoals:

`dt_cv_float_TCC7.2.1:`

```
{-1}  interpreted_data_type?[((range_floating_point(cv_base(t'))))]
        (dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
{-2}  Cpp_Type?(const(t'))
{-3}  cv(floating_point?)(const(t'))
{-4}  typ' = const(t')
------------------------------------------------------------------------
{1}   interpreted_data_type?[((range_floating_point(cv_base(typ'))))]
        (dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,
Applying propositional simplification,
we get 2 subgoals:

`dt_cv_float_TCC7.2.1.1:`

```
{-1}  uninterpreted_data_type?(dt_const[((range_floating_point(cv_base(t'))))]
                                 (v'(t'))`uidt)
{-2}  ∀ (d: (((range_floating_point(cv_base(t'))))), a: Address):
        valid?(uidt(dt_const[((range_floating_point(cv_base(t'))))](v'(t'))))
          (to_byte(dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
             (d, a),
           a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_const[((range_floating_point(cv_base(t'))))](v'(t'))))
          (l, a)
        ≡
        up?(from_byte(dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
             (l, a))
{-4}  ∀ (d: (((range_floating_point(cv_base(t'))))), a: Address):
        down(from_byte(dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
               (to_byte(dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
                  (d, a),
                a))
        = d
{-5}  Cpp_Type?(const(t'))
{-6}  cv(floating_point?)(const(t'))
{-7}  typ' = const(t')
------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(cv_base(typ'))))), a: Address):
        valid?(uidt(dt_const[((range_floating_point(cv_base(t'))))](v'(t'))))
          (to_byte(dt_const[((range_floating_point(cv_base(t'))))](v'(t')))
             (d, a),
           a)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Replacing using formula -8,
Expanding the definition of cv_base,
which is trivially true.
This completes the proof of `dt_cv_float_TCC7.2.1.1`.
`dt_cv_float_TCC7.2.1.2`:

{-1} uninterpreted_data_type?(dt_const[((range_floating_point(cv_base(t'))))] ($v'(t')$)`uidt)
{-2} ∀ (d: (((range_floating_point(cv_base(t'))))), a: Address):
valid?(uidt(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(to_byte(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(d, a),
a)
{-3} ∀ (l: list[Byte], a: Address):
valid?(uidt(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(l, a)
≡
up?(from_byte(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(l, a))
{-4} ∀ (d: (((range_floating_point(cv_base(t'))))), a: Address):
down(from_byte(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(to_byte(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(d, a),
a))
= d
{-5} Cpp_Type?(const(t'))
{-6} cv(floating_point?)(const(t'))
{-7} typ′ = const(t')

{1} ∀ (d: ((range_floating_point(cv_base(typ′)))), a: Address):
down(from_byte(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(to_byte(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))
(d, a),
a))
= d

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of cv_base,
Replacing using formula -8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC7.2.1.2`.
`dt_cv_float_TCC7.2.2`:

{-1} Cpp_Type?(const(t'))
{-2} cv(floating_point?)(const(t'))
{-3} typ′ = const(t')

{1} ∃ (x_1: (interpreted_data_type?[((range_floating_point(cv_base(t'))))])): TRUE
{2} interpreted_data_type?[((range_floating_point(cv_base(typ′))))]
(dt_const[((range_floating_point(cv_base(t'))))]($v'(t')$))

Using lemma dt_cv_float_TCC6,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC7.2.2`.

dt_cv_float_TCC7.2.3:

{-1}	Cpp_Type?(const(t'))
{-2}	cv(floating_point?)(const(t'))
{-3}	typ$'$ = const(t')
{1}	floating_point?(cv_base(t'))
{2}	interpreted_data_type?[((range_floating_point(cv_base(typ$'$))))] (dt_const[((range_floating_point(cv_base(t'))))](v'(t')))

Using lemma dt_cv_float_TCC5,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_float_TCC7.2.3.
Q.E.D.

C.44.91 Cpp_Types.dt_cv_float_TCC8

Terse proof for dt_cv_float_TCC8.
dt_cv_float_TCC8:

{1}	$\forall$ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = volatile(t) $\supset$ Cpp_Type?(t) $\wedge$ cv(floating_point?)(t)

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
dt_cv_float_TCC8.1:

{-1}	Cpp_Type?(typ$'$)
{-2}	cv(floating_point?)(typ$'$)
{-3}	typ$'$ = volatile(t')
{1}	Cpp_Type?(t')

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (t!2),
Expanding the definition of subterm,
Replacing using formula -4,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of dt_cv_float_TCC8.1.
dt_cv_float_TCC8.2:

{-1}	Cpp_Type?(typ$'$)
{-2}	cv(floating_point?)(typ$'$)
{-3}	typ$'$ = volatile(t')
{1}	cv(floating_point?)(t')

Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of dt_cv_float_TCC8.2.
Q.E.D.

C.44.92 Cpp_Types.dt_cv_float_TCC9

Terse proof for dt_cv_float_TCC9.

`dt_cv_float_TCC9:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_):
       typ = volatile(t) ⊃
        restrict
             [[Cpp_Type_, Cpp_Type_],
                [Cpp_Subtype(cv(floating_point?)), Cpp_Subtype(cv(floating_point?))], boolean]
             (≪)(t, typ)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of restrict,
Expanding the definition of ■,
Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC9`.
Q.E.D.

C.44.93 Cpp_Types.dt_cv_float_TCC10

Terse proof for `dt_cv_float_TCC10`.

`dt_cv_float_TCC10:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_):
       typ = volatile(t) ⊃ Cpp_Type?(t)
```

Repeatedly Skolemizing and flattening,
Using lemma dt_cv_float_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC10`.
Q.E.D.

C.44.94 Cpp_Types.dt_cv_float_TCC11

Terse proof for `dt_cv_float_TCC11`.

`dt_cv_float_TCC11:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_):
       typ = volatile(t) ⊃ floating_point?(cv_base(t))
```

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:

`dt_cv_float_TCC11.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(floating_point?)(typ′)
{-3}  typ′ = volatile(t′)
-------------------------------
{1}   cv(floating_point?)(t′)
{2}   floating_point?(cv_base(t′))
```

Using lemma dt_cv_float_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC11.1`.

`dt_cv_float_TCC11.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = volatile(t')
{1}	(floating_point? ⊆ interpreted?)
{2}	floating_point?(cv_base(t'))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_float_TCC11.2`.
Q.E.D.

C.44.95 Cpp_Types.dt_cv_float_TCC12

Terse proof for `dt_cv_float_TCC12`.

`dt_cv_float_TCC12:`

{1}	∀ (typ: Cpp_Subtype(cv(floating_point?)), t: Cpp_Type_): typ = volatile(t) ⊃ (∃ (x_1: (interpreted_data_type?[((range_floating_point(cv_base(t))))])): TRUE)

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (dt_floating_point(cv_base(t!1))),
we get 3 subgoals:

`dt_cv_float_TCC12.1:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = volatile(t')
{1}	TRUE

which is trivially true.
This completes the proof of `dt_cv_float_TCC12.1`.

`dt_cv_float_TCC12.2:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = volatile(t')
{1}	floating_point?(cv_base(t'))

Using lemma dt_cv_float_TCC11,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC12.2`.

`dt_cv_float_TCC12.3:`

{-1}	Cpp_Type?(typ′)
{-2}	cv(floating_point?)(typ′)
{-3}	typ′ = volatile(t')
{1}	Cpp_Type?(t')

Using lemma dt_cv_float_TCC8,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC12.3`.
Q.E.D.

C.44.96 Cpp_Types.dt_cv_float_TCC13

Terse proof for `dt_cv_float_TCC13`.

`dt_cv_float_TCC13:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?)),
          v:
            [typ1:
               {z: Cpp_Subtype(cv(floating_point?)) |
                     restrict
                       [[Cpp_Type_, Cpp_Type_],
                         [Cpp_Subtype(cv(floating_point?)),
                           Cpp_Subtype(cv(floating_point?))],
                         boolean]
                       (≪)(z, typ)} →
                 (interpreted_data_type?[((range_floating_point(cv_base(typ1))))])],
          t: Cpp_Type_):
       typ = volatile(t) ⊃
        (∀ (x: extended_real):
           range_floating_point(cv_base(t))(x) ≡
            range_floating_point(cv_base(typ))(x))
         ∧
         (∀ (x: extended_real):
            range_floating_point(cv_base(t))(x) ≡
             range_floating_point(cv_base(typ))(x))
          ∧
          (∀ (x1: [list[Byte], Address]):
             every[((range_floating_point(cv_base(t))))]
                  ((range_floating_point(cv_base(typ))))
                  (dt_volatile[((range_floating_point(cv_base(t))))](v(t))`from_byte
                         (x1)))
           ∧
           interpreted_data_type?[(range_floating_point(cv_base(typ)))]
                (dt_volatile[((range_floating_point(cv_base(t))))](v(t)))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of cv_base,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`dt_cv_float_TCC13.1:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(floating_point?)(typ′)
{-3}  typ′ = volatile(t′)
-------------------------------------------------------------------
 {1}  interpreted_data_type?[((range_floating_point(cv_base(typ′))))]
        (dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
```

Adding type constraints for dt_volatile[((range_floating_point(cv_base(t!1))))](v!1(t!1)),

we get 3 subgoals:

`dt_cv_float_TCC13.1.1:`

```
{-1}  interpreted_data_type?[((range_floating_point(cv_base(t'))))]
        (dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
{-2}  Cpp_Type?(typ')
{-3}  cv(floating_point?)(typ')
{-4}  typ' = volatile(t')
------------------------------------------------------------------
{1}   interpreted_data_type?[((range_floating_point(cv_base(typ'))))]
        (dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
```

Expanding the definition of interpreted_data_type?,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

`dt_cv_float_TCC13.1.1.1:`

```
{-1}  uninterpreted_data_type?(dt_volatile[((range_floating_point(cv_base(t'))))]
                                 (v'(t'))`uidt)
{-2}  ∀ (d: (((range_floating_point(cv_base(t'))))), a: Address):
        valid?(uidt(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t'))))
          (to_byte(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
             (d, a),
           a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t'))))
          (l, a)
         ≡
         up?(from_byte(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
             (l, a))
{-4}  ∀ (d: (((range_floating_point(cv_base(t'))))), a: Address):
        down(from_byte(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
             (to_byte(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
                (d, a),
              a))
        = d
{-5}  Cpp_Type?(typ')
{-6}  cv(floating_point?)(typ')
{-7}  typ' = volatile(t')
------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(cv_base(typ'))))), a: Address):
        valid?(uidt(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t'))))
          (to_byte(dt_volatile[((range_floating_point(cv_base(t'))))](v'(t')))
             (d, a),
           a)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -8,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_float_TCC13.1.1.1`.

`dt_cv_float_TCC13.1.1.2:`

```
{-1}  uninterpreted_data_type?(dt_volatile[((range_floating_point(cv_base(t′))))]
                                          (v′(t′))`uidt)
{-2}  ∀ (d: (((range_floating_point(cv_base(t′))))), a: Address):
        valid?(uidt(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′))))
            (to_byte(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
                (d, a),
              a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′))))
            (l, a)
          ≡
          up?(from_byte(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
                (l, a))
{-4}  ∀ (d: (((range_floating_point(cv_base(t′))))), a: Address):
        down(from_byte(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
                (to_byte(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
                    (d, a),
                  a))
          = d
{-5}  Cpp_Type?(typ′)
{-6}  cv(floating_point?)(typ′)
{-7}  typ′ = volatile(t′)
-------------------------------------------------------------------------
{1}   ∀ (d: ((range_floating_point(cv_base(typ′)))), a: Address):
        down(from_byte(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
                (to_byte(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
                    (d, a),
                  a))
          = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Replacing using formula -8,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `dt_cv_float_TCC13.1.1.2`.

`dt_cv_float_TCC13.1.2:`

```
{-1}  Cpp_Type?(typ′)
{-2}  cv(floating_point?)(typ′)
{-3}  typ′ = volatile(t′)
-------------------------------------------------------------------------
{1}   ∃ (x1: (interpreted_data_type?[((range_floating_point(cv_base(t′))))])): TRUE
{2}   interpreted_data_type?[((range_floating_point(cv_base(typ′))))]
          (dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))
```

Using lemma dt_cv_float_TCC12,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `dt_cv_float_TCC13.1.2`.

`dt_cv_float_TCC13.1.3:`

{-1} Cpp_Type?(typ′)
{-2} cv(floating_point?)(typ′)
{-3} typ′ = volatile(t′)

{1} floating_point?(cv_base(t′))
{2} interpreted_data_type?[((range_floating_point(cv_base(typ′))))]
(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′)))

Using lemma dt_cv_float_TCC11,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC13.1.3`.
`dt_cv_float_TCC13.2:`

{-1} Cpp_Type?(typ′)
{-2} cv(floating_point?)(typ′)
{-3} typ′ = volatile(t′)

{1} ∀ (x1: [list[Byte], Address]):
every[((range_floating_point(cv_base(t′))))]
((range_floating_point(cv_base(typ′))))
(dt_volatile[((range_floating_point(cv_base(t′))))](v′(t′))‘from_byte
(x1))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of cv_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC13.2`.
`dt_cv_float_TCC13.3:`

{-1} Cpp_Type?(typ′)
{-2} cv(floating_point?)(typ′)
{-3} typ′ = volatile(t′)

{1} ∀ (x: extended_real):
range_floating_point(cv_base(t′))(x) ≡
range_floating_point(cv_base(typ(typ′)))(x)

Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC13.3`.
Q.E.D.

C.44.97 Cpp_Types.dt_cv_float_TCC14

Terse proof for `dt_cv_float_TCC14`.
`dt_cv_float_TCC14:`

{1} ∀ (typ: Cpp_Subtype(cv(floating_point?))):
¬ volatile?(typ) ∧ ¬ const?(typ) ⊃ floating_point?(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `dt_cv_float_TCC14`.
Q.E.D.

C.44.98 Cpp_Types.dt_cv_float_TCC15

Terse proof for `dt_cv_float_TCC15`.
`dt_cv_float_TCC15:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(floating_point?))):
       ¬ volatile?(typ) ∧ ¬ const?(typ) ⊃
        (∀ (x: extended_real):
             range_floating_point(typ)(x) ≡ range_floating_point(cv_base(typ))(x))
         ∧
         (∀ (x: extended_real):
              range_floating_point(typ)(x) ≡ range_floating_point(cv_base(typ))(x))
          ∧
          (∀ (x1: [list[Byte], Address]):
               every[((range_floating_point(typ)))]
                    ((range_floating_point(cv_base(typ))))
                    (dt_floating_point(typ)`from_byte(x1)))
           ∧
           interpreted_data_type?[(range_floating_point(cv_base(typ)))]
                (dt_floating_point(typ))
```

Repeatedly Skolemizing and flattening,
Using lemma cv_base_result2,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`dt_cv_float_TCC15.1:`

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(typ′)
{-3}  cv(floating_point?)(typ′)
-------------------------------
{1}   volatile?(typ′)
{2}   const?(typ′)
{3}   interpreted_data_type?[((range_floating_point(cv_base(typ′))))]
          (dt_floating_point(typ′))
```

Replacing using formula -1,
Adding type constraints for dt_floating_point(cv_base(typ!1)),
Rewriting using pod_is_interpreted_data, matching in *,
This completes the proof of `dt_cv_float_TCC15.1`.
`dt_cv_float_TCC15.2:`

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(typ′)
{-3}  cv(floating_point?)(typ′)
-------------------------------
{1}   volatile?(typ′)
{2}   const?(typ′)
{3}   ∀ (x1: [list[Byte], Address]):
        every[((range_floating_point(typ′)))]
             ((range_floating_point(cv_base(typ′))))(dt_floating_point(typ′)`from_byte(x1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC15.2`.
`dt_cv_float_TCC15.3`:

```
{-1}  cv_base(typ′) = typ′
{-2}  Cpp_Type?(typ′)
{-3}  cv(floating_point?)(typ′)
------------------------------------------------------------------------
{1}   volatile?(typ′)
{2}   const?(typ′)
{3}   ∀ (x: extended_real):
        range_floating_point(typ′)(x) ≡ range_floating_point(cv_base(typ′))(x)
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `dt_cv_float_TCC15.3`.
Q.E.D.

C.44.99 Cpp_Types.uidt_TCC1

Terse proof for `uidt_TCC1`.
`uidt_TCC1`:

```
------------------------------------------------------------------------
{1}   ∀ (typ: Cpp_Type, r: Cpp_Type_, a: list[Cpp_Type_], v: bool):
        typ = function_type(r, a, v) ⊃ function?(typ)
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC1`.
Q.E.D.

C.44.100 Cpp_Types.uidt_TCC2

Terse proof for `uidt_TCC2`.
`uidt_TCC2`:

```
------------------------------------------------------------------------
{1}   ∀ (typ: Cpp_Type, t: Cpp_Type_): typ = pointer(t) ⊃ cv(pointer?)(typ)
```

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC2`.
Q.E.D.

C.44.101 Cpp_Types.uidt_TCC3

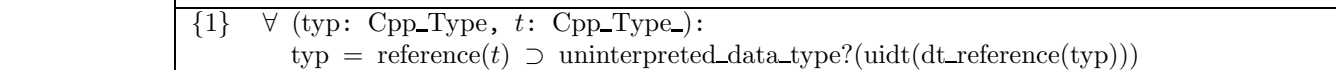
Terse proof for `uidt_TCC3`.
`uidt_TCC3`:

```
------------------------------------------------------------------------
{1}   ∀ (typ: Cpp_Type, t: Cpp_Type_):
        typ = pointer(t) ⊃ uninterpreted_data_type?(uidt(dt_cv_pointer(typ)))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_pointer(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC3`.
Q.E.D.

C.44.102 Cpp_Types.uidt_TCC4

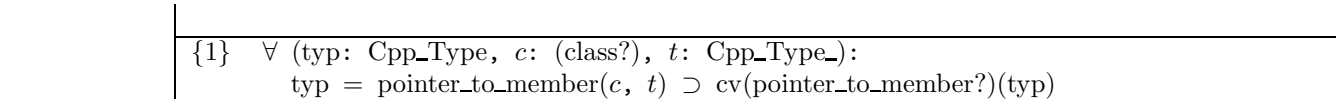
Terse proof for `uidt_TCC4`.
`uidt_TCC4:`



Repeatedly Skolemizing and flattening,
Adding type constraints for dt_reference(typ!1),
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC4`.
Q.E.D.

C.44.103 Cpp_Types.uidt_TCC5

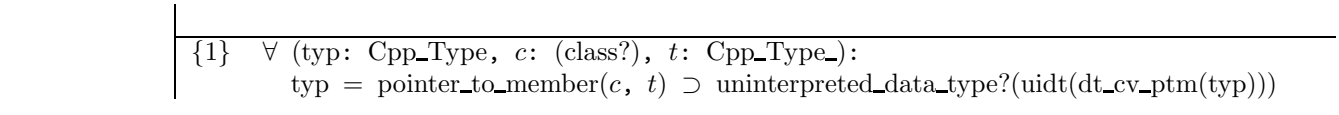
Terse proof for `uidt_TCC5`.
`uidt_TCC5:`



Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC5`.
Q.E.D.

C.44.104 Cpp_Types.uidt_TCC6

Terse proof for `uidt_TCC6`.
`uidt_TCC6:`



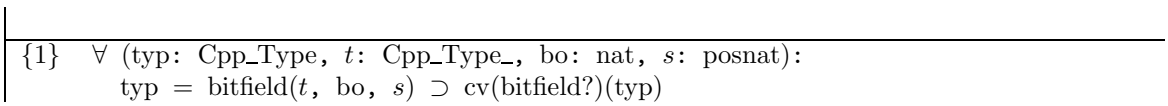
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Adding type constraints for dt_cv_ptm(pointer_to_member(c!1, t!1)),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC6`.

Q.E.D.

C.44.105 Cpp_Types.uidt_TCC7

Terse proof for `uidt_TCC7`.

`uidt_TCC7`:

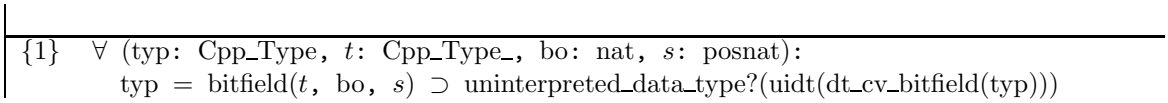


Repeatedly Skolemizing and flattening,
Replacing using formula -4,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC7`.
Q.E.D.

C.44.106 Cpp_Types.uidt_TCC8

Terse proof for `uidt_TCC8`.

`uidt_TCC8`:

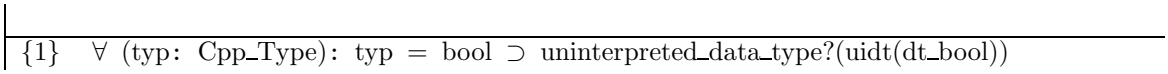


Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_bitfield(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC8`.
Q.E.D.

C.44.107 Cpp_Types.uidt_TCC9

Terse proof for `uidt_TCC9`.

`uidt_TCC9`:



Repeatedly Skolemizing and flattening,
Adding type constraints for dt_bool,
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC9`.
Q.E.D.

C.44.108 Cpp_Types.uidt_TCC10

Terse proof for `uidt_TCC10`.

uidt_TCC10:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
t = bool ∧ typ = const(t) ⊃ uninterpreted_data_type?(uidt(dt_bool))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_bool,
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC10.
Q.E.D.

C.44.109 Cpp_Types.uidt_TCC11

Terse proof for uidt_TCC11.

uidt_TCC11:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃ (∀ (t_1: Cpp_Type_): t = pointer(t_1) ⊃ cv(pointer?)(typ))

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_TCC11.
Q.E.D.

C.44.110 Cpp_Types.uidt_TCC12

Terse proof for uidt_TCC12.

uidt_TCC12:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃
(∀ (t_1: Cpp_Type_):
t = pointer(t_1) ⊃ uninterpreted_data_type?(uidt(dt_cv_pointer(typ))))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_pointer(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC12.
Q.E.D.

C.44.111 Cpp_Types.uidt_TCC13

Terse proof for uidt_TCC13.

uidt_TCC13:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃
(∀ (c: (class?), t_1: Cpp_Type_):
t = pointer_to_member(c, t_1) ⊃ cv(pointer_to_member?)(typ))

Repeatedly Skolemizing and flattening,
Hiding formulas: -2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_TCC13.
Q.E.D.

C.44.112 Cpp_Types.uidt_TCC14

Terse proof for uidt_TCC14.

uidt_TCC14:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃
(∀ (c: (class?), t_1: Cpp_Type_):
t = pointer_to_member(c, t_1) ⊃ uninterpreted_data_type?(uidt(dt_cv_ptm(typ))))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_ptm(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC14.
Q.E.D.

C.44.113 Cpp_Types.uidt_TCC15

Terse proof for uidt_TCC15.

uidt_TCC15:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃
(∀ (t_1: Cpp_Type_, bo: nat, s: posnat):
t = bitfield(t_1, bo, s) ⊃ cv(bitfield?)(typ))

Repeatedly Skolemizing and flattening,
Hiding formulas: -3,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_TCC15.
Q.E.D.

C.44.114 Cpp_Types.uidt_TCC16

Terse proof for uidt_TCC16.

uidt_TCC16:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = const(t) ⊃
(∀ (t_1: Cpp_Type_, bo: nat, s: posnat):
t = bitfield(t_1, bo, s) ⊃ uninterpreted_data_type?(uidt(dt_cv_bitfield(typ))))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_bitfield(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC16.
Q.E.D.

C.44.115 Cpp_Types.uidt_TCC17

Terse proof for uidt_TCC17.

uidt_TCC17:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
floating_point?(t) ∧
¬ volatile?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
⊃ cv(floating_point?)(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_TCC17.
Q.E.D.

C.44.116 Cpp_Types.uidt_TCC18

Terse proof for uidt_TCC18.

uidt_TCC18:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
floating_point?(t) ∧
¬ volatile?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
⊃ uninterpreted_data_type?(uidt(dt_cv_float(typ)))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_float(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC18.
Q.E.D.

C.44.117 Cpp_Types.uidt_TCC19

Terse proof for `uidt_TCC19`.

`uidt_TCC19:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
non_bool_integral_enum?(t) ∧
¬ volatile?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
⊃ cv(non_bool_integral_enum?)(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC19`.
Q.E.D.

C.44.118 Cpp_Types.uidt_TCC20

Terse proof for `uidt_TCC20`.

`uidt_TCC20:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
non_bool_integral_enum?(t) ∧
¬ volatile?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
⊃ uninterpreted_data_type?(uidt(dt(typ)))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt(typ!1),
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC20`.
Q.E.D.

C.44.119 Cpp_Types.uidt_TCC21

Terse proof for `uidt_TCC21`.

`uidt_TCC21:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
¬ volatile?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
⊃ ¬ (floating_point?(t) ∧ non_bool_integral_enum?(t))

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC21`.

Q.E.D.

C.44.120 Cpp_Types.uidt_TCC22

Terse proof for `uidt_TCC22`.

`uidt_TCC22:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
¬ volatile?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = const(t)
⊃ floating_point?(t) ∨ non_bool_integral_enum?(t)

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Instantiating the top quantifier in -1 with the terms: (typ!1),
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Keeping (-3 -17 -21 -23 -24 -25 -27 -29 1 2 3 4 5 6 7) and hiding *,
Replacing using formula -8,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC22`.
Q.E.D.

C.44.121 Cpp_Types.uidt_TCC23

Terse proof for `uidt_TCC23`.

`uidt_TCC23:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
t = bool ∧ typ = volatile(t) ⊃ uninterpreted_data_type?(uidt(dt_bool))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_bool,
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC23`.
Q.E.D.

C.44.122 Cpp_Types.uidt_TCC24

Terse proof for `uidt_TCC24`.

`uidt_TCC24:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = volatile(t) ⊃ (∀ (t_1: Cpp_Type_): t = pointer(t_1) ⊃ cv(pointer?)(typ))

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC24`.

Q.E.D.

C.44.123 Cpp_Types.uidt_TCC25

Terse proof for `uidt_TCC25`.

`uidt_TCC25`:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = volatile(t) ⊃
(∀ (t_1: Cpp_Type_):
t = pointer(t_1) ⊃ uninterpreted_data_type?(uidt(dt_cv_pointer(typ))))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_pointer(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC25`.
Q.E.D.

C.44.124 Cpp_Types.uidt_TCC26

Terse proof for `uidt_TCC26`.

`uidt_TCC26`:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = volatile(t) ⊃
(∀ (c: (class?), t_1: Cpp_Type_):
t = pointer_to_member(c, t_1) ⊃ cv(pointer_to_member?)(typ))

Repeatedly Skolemizing and flattening,
Hiding formulas: -2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC26`.
Q.E.D.

C.44.125 Cpp_Types.uidt_TCC27

Terse proof for `uidt_TCC27`.

`uidt_TCC27`:

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
typ = volatile(t) ⊃
(∀ (c: (class?), t_1: Cpp_Type_):
t = pointer_to_member(c, t_1) ⊃ uninterpreted_data_type?(uidt(dt_cv_ptm(typ))))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_ptm(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC27`.
Q.E.D.

C.44.126 Cpp_Types.uidt_TCC28

Terse proof for `uidt_TCC28`.

`uidt_TCC28`:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
       typ = volatile(t) ⊃
        (∀ (t1: Cpp_Type_, bo: nat, s: posnat):
             t = bitfield(t1, bo, s) ⊃ cv(bitfield?)(typ))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -3,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC28`.
Q.E.D.

C.44.127 Cpp_Types.uidt_TCC29

Terse proof for `uidt_TCC29`.

`uidt_TCC29`:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
       typ = volatile(t) ⊃
        (∀ (t1: Cpp_Type_, bo: nat, s: posnat):
             t = bitfield(t1, bo, s) ⊃ uninterpreted_data_type?(uidt(dt_cv_bitfield(typ))))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_bitfield(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC29`.
Q.E.D.

C.44.128 Cpp_Types.uidt_TCC30

Terse proof for `uidt_TCC30`.

`uidt_TCC30`:

```
{1}  ∀ (typ: Cpp_Type, t: Cpp_Type_):
       floating_point?(t) ∧
        ¬ bitfield?(t) ∧
         ¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
        ⊃ cv(floating_point?)(typ)
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC30`.
Q.E.D.

C.44.129 Cpp_Types.uidt_TCC31

Terse proof for `uidt_TCC31`.

`uidt_TCC31:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
floating_point?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
⊃ uninterpreted_data_type?(uidt(dt_cv_float(typ)))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_float(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC31`.
Q.E.D.

C.44.130 Cpp_Types.uidt_TCC32

Terse proof for `uidt_TCC32`.

`uidt_TCC32:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
non_bool_integral_enum?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
⊃ cv(non_bool_integral_enum?)(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC32`.
Q.E.D.

C.44.131 Cpp_Types.uidt_TCC33

Terse proof for `uidt_TCC33`.

`uidt_TCC33:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
non_bool_integral_enum?(t) ∧
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
⊃ uninterpreted_data_type?(uidt(dt(typ)))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt(typ!1),
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `uidt_TCC33`.
Q.E.D.

C.44.132 Cpp_Types.uidt_TCC34

Terse proof for `uidt_TCC34`.

`uidt_TCC34:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
⊃ ¬ (floating_point?(t) ∧ non_bool_integral_enum?(t))

Repeatedly Skolemizing and flattening,

Keeping (-3 -4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `uidt_TCC34`.

Q.E.D.

C.44.133 Cpp_Types.uidt_TCC35

Terse proof for `uidt_TCC35`.

`uidt_TCC35:`

{1} ∀ (typ: Cpp_Type, t: Cpp_Type_):
¬ bitfield?(t) ∧
¬ pointer_to_member?(t) ∧ ¬ pointer?(t) ∧ ¬ bool?(t) ∧ typ = volatile(t)
⊃ floating_point?(t) ∨ non_bool_integral_enum?(t)

Repeatedly Skolemizing and flattening,

Expanding the definition of Cpp_Type?,

Instantiating the top quantifier in -1 with the terms: (typ!1),

Expanding the definition of subterm,

Applying disjunctive simplification to flatten sequent,

Replacing using formula -29,

Keeping (-3 -17 -20 -22 -23 -24 -25 -27 1 2 3 4 5 6) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `uidt_TCC35`.

Q.E.D.

C.44.134 Cpp_Types.uidt_TCC36

Terse proof for `uidt_TCC36`.

uidt_TCC36:

{1} ∀ (typ: Cpp_Type):
floating_point?(typ) ∧
¬ volatile?(typ) ∧
¬ const?(typ) ∧
¬ bool?(typ) ∧
¬ bitfield?(typ) ∧
¬ pointer_to_member?(typ) ∧
¬ reference?(typ) ∧
¬ pointer?(typ) ∧
¬ union?(typ) ∧
¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
⊃ cv(floating_point?)(typ)

Repeatedly Skolemizing and flattening,
Keeping (-2 13) and hiding *,
Expanding the definition of cv,
Expanding the definition of c,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC36.
Q.E.D.

C.44.135 Cpp_Types.uidt_TCC37

Terse proof for uidt_TCC37.

uidt_TCC37:

{1} ∀ (typ: Cpp_Type):
floating_point?(typ) ∧
¬ volatile?(typ) ∧
¬ const?(typ) ∧
¬ bool?(typ) ∧
¬ bitfield?(typ) ∧
¬ pointer_to_member?(typ) ∧
¬ reference?(typ) ∧
¬ pointer?(typ) ∧
¬ union?(typ) ∧
¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
⊃ uninterpreted_data_type?(uidt(dt_cv_float(typ)))

Repeatedly Skolemizing and flattening,
Adding type constraints for dt_cv_float(typ!1),
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC37.
Q.E.D.

C.44.136 Cpp_Types.uidt_TCC38

Terse proof for uidt_TCC38.

uidt_TCC38:

```
{1}   ∀ (typ: Cpp_Type):
        non_bool_integral_enum?(typ) ∧
         ¬ volatile?(typ) ∧
          ¬ const?(typ) ∧
           ¬ bool?(typ) ∧
            ¬ bitfield?(typ) ∧
             ¬ pointer_to_member?(typ) ∧
              ¬ reference?(typ) ∧
               ¬ pointer?(typ) ∧
                ¬ union?(typ) ∧
                 ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
         ⊃ cv(non_bool_integral_enum?)(typ)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of cv,
Expanding the definition of c,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC38.
Q.E.D.

C.44.137 Cpp_Types.uidt_TCC39

Terse proof for uidt_TCC39.

uidt_TCC39:

```
{1}   ∀ (typ: Cpp_Type):
        non_bool_integral_enum?(typ) ∧
         ¬ volatile?(typ) ∧
          ¬ const?(typ) ∧
           ¬ bool?(typ) ∧
            ¬ bitfield?(typ) ∧
             ¬ pointer_to_member?(typ) ∧
              ¬ reference?(typ) ∧
               ¬ pointer?(typ) ∧
                ¬ union?(typ) ∧
                 ¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
         ⊃ uninterpreted_data_type?(uidt(dt(typ)))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for dt(typ!1),
Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_TCC39.
Q.E.D.

C.44.138 Cpp_Types.uidt_TCC40

Terse proof for uidt_TCC40.

`uidt_TCC40:`

{1} ∀ (typ: Cpp_Type):
¬ volatile?(typ) ∧
¬ const?(typ) ∧
¬ bool?(typ) ∧
¬ bitfield?(typ) ∧
¬ pointer_to_member?(typ) ∧
¬ reference?(typ) ∧
¬ pointer?(typ) ∧
¬ union?(typ) ∧
¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
⊃ ¬ (floating_point?(typ) ∧ non_bool_integral_enum?(typ))

Repeatedly Skolemizing and flattening,
Keeping (-2 -3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC40`.
Q.E.D.

C.44.139 Cpp_Types.uidt_TCC41

Terse proof for `uidt_TCC41`.

`uidt_TCC41:`

{1} ∀ (typ: Cpp_Type):
¬ volatile?(typ) ∧
¬ const?(typ) ∧
¬ bool?(typ) ∧
¬ bitfield?(typ) ∧
¬ pointer_to_member?(typ) ∧
¬ reference?(typ) ∧
¬ pointer?(typ) ∧
¬ union?(typ) ∧
¬ class?(typ) ∧ ¬ function?(typ) ∧ ¬ array?(typ) ∧ ¬ void?(typ)
⊃ floating_point?(typ) ∨ non_bool_integral_enum?(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_TCC41`.
Q.E.D.

C.44.140 Cpp_Types.mod_float_TCC1

Terse proof for `mod_float_TCC1`.

`mod_float_TCC1:`

{1} ∃ (x: [Cpp_Subtype(cv(floating_point?)) → [extended_real → extended_real]]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (t: Cpp_Subtype(cv(floating_point?))): λ (e: extended_real): e),

which is trivially true.
This completes the proof of `mod_float_TCC1`.
Q.E.D.

C.44.141 Cpp_Types.value_bitmask_TCC1

Terse proof for `value_bitmask_TCC1`.
`value_bitmask_TCC1:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(non_bool_integral_enum?))):
       ¬ (bool?(typ) ∨
            float?(typ) ∨
              double?(typ) ∨
                longdouble?(typ) ∨
                  void?(typ) ∨
                    array?(typ) ∨
                      function?(typ) ∨
                        pointer?(typ) ∨
                          reference?(typ) ∨
                            class?(typ) ∨ union?(typ) ∨ bitfield?(typ) ∨ pointer_to_member?(typ))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of cv,
Expanding the definition of c,
Expanding the definition of v,
Expanding the definition of non_bool_integral_enum?,
Expanding the definition of non_bool_integral?,
Expanding the definition of unsigned_integer?,
Expanding the definition of signed_integer?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `value_bitmask_TCC1`.
Q.E.D.

C.45 Proofs for Cpp_Verification (cpp-verification.pvs)

This theory contains no provable formal statements.

C.46 Proofs for Cpp_array (types.pvs)

C.46.1 Cpp_array.uidt_array_TCC1

Terse proof for `uidt_array_TCC1`.
`uidt_array_TCC1:`

```
{1}  ∃ (x: ⌊Cpp_Subtype(array?) → (uninterpreted_data_type?)⌋): TRUE
```

Instantiating the top quantifier in 1 with the terms: (λ (c: Cpp_Subtype(array?)): choose(uninterpreted_data_type?)),
we get 2 subgoals:

uidt_array_TCC1.1:

|
{1} TRUE

which is trivially true.
This completes the proof of uidt_array_TCC1.1.
uidt_array_TCC1.2:

|
{1} ∀ (c: Cpp_Subtype(array?)): nonempty?[Uninterpreted_data_type](uninterpreted_data_type?)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in -3 with the terms: ((# size := 0, valid? := LAMBDA (l: list[Byte], a: Address): FALSE #)),
Repeatedly Skolemizing and flattening,
This completes the proof of uidt_array_TCC1.2.
Q.E.D.

C.47 Proofs for Cpp_bitfield (types.pvs)

C.47.1 Cpp_bitfield.range_bitfield_TCC1

Terse proof for range_bitfield_TCC1.
range_bitfield_TCC1:

|
{1} ∃ (x: [Cpp_Subtype(bitfield?) → non_empty_finite_set[int]]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (b: Cpp_Subtype(bitfield?)): λ (i: int): i = 0),
we get 2 subgoals:
range_bitfield_TCC1.1:

|
{1} TRUE

which is trivially true.
This completes the proof of range_bitfield_TCC1.1.
range_bitfield_TCC1.2:

|
{1} ∀ (b: Cpp_Subtype(bitfield?)):
is_finite[int](λ (i: int): i = 0) ∧ ¬ empty?[int](λ (i: int): i = 0)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (i: int): i = 0)): 0),
which is trivially true.
This completes the proof of range_bitfield_TCC1.2.
Q.E.D.

C.47.2 Cpp_bitfield.bitfield_range_underlying_TCC1

Terse proof for `bitfield_range_underlying_TCC1`.

`bitfield_range_underlying_TCC1:`

{1} ∀ (*b*: Cpp_Subtype(bitfield?)):
array?(*b*) ∨
pointer?(*b*) ∨
reference?(*b*) ∨
bitfield?(*b*) ∨ enum?(*b*) ∨ pointer_to_member?(*b*) ∨ const?(*b*) ∨ volatile?(*b*)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `bitfield_range_underlying_TCC1`.
Q.E.D.

C.47.3 Cpp_bitfield.bitfield_range_underlying_TCC2

Terse proof for `bitfield_range_underlying_TCC2`.

`bitfield_range_underlying_TCC2:`

{1} ∀ (*b*: Cpp_Subtype(bitfield?)): enum?(typ(*b*)) ⊃ Cpp_Type?(typ(*b*))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bitfield_range_underlying_TCC2`.
Q.E.D.

C.47.4 Cpp_bitfield.bitfield_range_underlying_TCC3

Terse proof for `bitfield_range_underlying_TCC3`.

`bitfield_range_underlying_TCC3:`

{1} ∀ (*b*: Cpp_Subtype(bitfield?)): ¬ enum?(typ(*b*)) ⊃ non_bool_integral?(typ(*b*))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Keeping (-16 -29 1 2) and hiding *,
Expanding the definition of bitfield_underlying_integral_or_enum_type?,
Expanding the definition of non_bool_integral_enum?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bitfield_range_underlying_TCC3`.
Q.E.D.

C.47.5 Cpp_bitfield.dt_bitfield_TCC1

Terse proof for `dt_bitfield_TCC1`.

`dt_bitfield_TCC1:`

{1} ∃ (x_1: [b: Cpp_Subtype(bitfield?) → (interpreted_data_type?[((range_bitfield(b)))])]):
TRUE

Instantiating the top quantifier in 1 with the terms: (λ (b: Cpp_Subtype(bitfield?)): choose(interpreted_data_type?[((range_bitfield(b)))])),

we get 2 subgoals:

`dt_bitfield_TCC1.1:`

{1} TRUE

which is trivially true.

This completes the proof of `dt_bitfield_TCC1.1`.

`dt_bitfield_TCC1.2:`

{1} ∀ (b: Cpp_Subtype(bitfield?)):
nonempty?[Interpreted_data_type[((range_bitfield(b)))]]
(interpreted_data_type?[((range_bitfield(b)))])

Repeatedly Skolemizing and flattening,

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Expanding the definition of member,

Using lemma dt_bitfield_exists,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `dt_bitfield_TCC1.2`.

Q.E.D.

C.48 Proofs for Cpp_bool (types.pvs)

C.48.1 Cpp_bool.range_bool_TCC1

Terse proof for `range_bool_TCC1`.

`range_bool_TCC1:`

{1} is_finite[bool](fullset[bool]) ∧ ¬ empty?[bool](fullset[bool])

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in 1 with the terms: (2 LAMBDA (b: (fullset[bool])): IF b THEN 0 ELSE 1 ENDIF),

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `range_bool_TCC1`.

Q.E.D.

C.48.2 Cpp_bool.dt_bool_TCC1

Terse proof for `dt_bool_TCC1`.
`dt_bool_TCC1:`

```
{1}   ∃ (x: (pod_data_type?[bool])): TRUE
```

Rewriting using dt_bool_exists, matching in *,
This completes the proof of `dt_bool_TCC1`.
Q.E.D.

C.49 Proofs for Cpp_char (types.pvs)

C.49.1 Cpp_char.dt_char_TCC1

Terse proof for `dt_char_TCC1`.
`dt_char_TCC1:`

```
{1}   ∃ (x_1: (pod_data_type?[(range_char)])): TRUE
```

Rewriting using dt_char_exists, matching in *,
This completes the proof of `dt_char_TCC1`.
Q.E.D.

C.49.2 Cpp_char.value_bitmask_char_TCC1

Terse proof for `value_bitmask_char_TCC1`.
`value_bitmask_char_TCC1:`

```
{1}   ∃ (x_1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_char))}): TRUE
```

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_char_TCC1`.
Q.E.D.

C.49.3 Cpp_char.min_char

Terse proof for `min_char`.
`min_char:`

```
{1}   min(range_char) ≤ 0
```

Using lemma char_is_uchar_or_schar,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`min_char.1:`

```
{-1}   range_char = range_schar
{1}    min(range_char) ≤ 0
```

Replacing using formula -1,

Using lemma min_schar,
Using lemma dt_schar_bits,
Using lemma min_bits_per_byte,
Using lemma bits_per_byte_minimum,
Rewriting using dt_schar_size, matching in *,
Rewriting using dt_uchar_size, matching in *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `min_char.1`.
`min_char.2:`

{-1}	range_char = extend[int, nat, bool, FALSE](range_uchar)
{1}	min(range_char) ≤ 0

Using lemma min_bits_per_byte,
Using lemma bits_per_byte_minimum,
Using lemma binary_range_uchar,
Using lemma dt_uchar_bits,
Rewriting using dt_uchar_size, matching in *,
Using lemma expt_ge1,
we get 2 subgoals:
`min_char.2.1:`

{-1}	2 ^ (max_value_bits_uchar − 8) ≥ 1
{-2}	max_value_bits_uchar = 1 × bits_per_byte
{-3}	range_uchar(0) ≡ 0 ≤ max(range_uchar)
{-4}	bits_per_byte ≥ min_bits_per_byte
{-5}	min_bits_per_byte = 8
{-6}	range_char = extend[int, nat, bool, FALSE](range_uchar)
{1}	min(range_char) ≤ 0

Trying repeated skolemization, instantiation, and if-lifting,
Keeping (-3 1) and hiding *,
Adding type constraints for min(extend[int, nat, bool, FALSE](range_uchar)),
Instantiating the top quantifier in -2 with the terms: (0),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `min_char.2.1`.
`min_char.2.2:`

{-1}	max_value_bits_uchar = 1 × bits_per_byte
{-2}	range_uchar(0) ≡ 0 ≤ max(range_uchar)
{-3}	bits_per_byte ≥ min_bits_per_byte
{-4}	min_bits_per_byte = 8
{-5}	range_char = extend[int, nat, bool, FALSE](range_uchar)
{1}	max_value_bits_uchar − 8 ≥ 0
{2}	min(range_char) ≤ 0

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `min_char.2.2`.
Q.E.D.

C.49.4 Cpp_char.max_char

Terse proof for `max_char`.

`max_char:`

{1}	max(range_char) $\geq$ 127

Using lemma char_is_uchar_or_schar,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`max_char.1:`

{-1}	range_char = range_schar
{1}	max(range_char) $\geq$ 127

Replacing using formula -1,
Using lemma max_schar,
Using lemma dt_schar_bits,
Using lemma min_bits_per_byte,
Using lemma bits_per_byte_minimum,
Rewriting using dt_schar_size, matching in *,
Rewriting using dt_uchar_size, matching in *,
Using lemma expt_ge1,
we get 2 subgoals:
`max_char.1.1:`

{-1}	2 ^ (max_value_bits_schar − 7) $\geq$ 1
{-2}	bits_per_byte $\geq$ min_bits_per_byte
{-3}	min_bits_per_byte = 8
{-4}	max_value_bits_schar $\geq$ 1 × bits_per_byte − 1
{-5}	max(range_schar) $\geq$ (2 ^ max_value_bits_schar) − 1
{-6}	range_char = range_schar
{1}	max(range_schar) $\geq$ 127

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `max_char.1.1`.
`max_char.1.2:`

{-1}	bits_per_byte $\geq$ min_bits_per_byte
{-2}	min_bits_per_byte = 8
{-3}	max_value_bits_schar $\geq$ 1 × bits_per_byte − 1
{-4}	max(range_schar) $\geq$ (2 ^ max_value_bits_schar) − 1
{-5}	range_char = range_schar
{1}	max_value_bits_schar − 7 $\geq$ 0
{2}	max(range_schar) $\geq$ 127

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `max_char.1.2`.
`max_char.2:`

{-1}	range_char = extend[int, nat, bool, FALSE](range_uchar)
{1}	max(range_char) $\geq$ 127

Using lemma max_uchar,
Using lemma dt_uchar_bits,
Using lemma min_bits_per_byte,
Using lemma bits_per_byte_minimum,
Rewriting using dt_uchar_size, matching in *,
Using lemma expt_ge1,

we get 2 subgoals:
`max_char.2.1:`

{-1}	$2 \hat{} (\text{max\_value\_bits\_uchar} - 8) \geq 1$
{-2}	$\text{bits\_per\_byte} \geq \text{min\_bits\_per\_byte}$
{-3}	$\text{min\_bits\_per\_byte} = 8$
{-4}	$\text{max\_value\_bits\_uchar} = 1 \times \text{bits\_per\_byte}$
{-5}	$\max(\text{range\_uchar}) \geq (2 \hat{} \text{max\_value\_bits\_uchar}) - 1$
{-6}	$\text{range\_char} = \text{extend}[\text{int, nat, bool, FALSE}](\text{range\_uchar})$
{1}	$\max(\text{range\_char}) \geq 127$

Replacing using formula -6,
Trying repeated skolemization, instantiation, and if-lifting,
Adding type constraints for max(extend[int, nat, bool, FALSE](range_uchar)),
Instantiating the top quantifier in -2 with the terms: (127),
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using binary_range_uchar, matching in * where n gets 127,
This completes the proof of `max_char.2.1`.
`max_char.2.2:`

{-1}	$\text{bits\_per\_byte} \geq \text{min\_bits\_per\_byte}$
{-2}	$\text{min\_bits\_per\_byte} = 8$
{-3}	$\text{max\_value\_bits\_uchar} = 1 \times \text{bits\_per\_byte}$
{-4}	$\max(\text{range\_uchar}) \geq (2 \hat{} \text{max\_value\_bits\_uchar}) - 1$
{-5}	$\text{range\_char} = \text{extend}[\text{int, nat, bool, FALSE}](\text{range\_uchar})$
{1}	$\text{max\_value\_bits\_uchar} - 8 \geq 0$
{2}	$\max(\text{range\_char}) \geq 127$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `max_char.2.2`.
Q.E.D.

C.49.5 Cpp_char.char_binary

Terse proof for `char_binary`.
`char_binary:`

{1}	$\forall (i\text{: int}): \text{range\_char}(i) \equiv (\min(\text{range\_char}) \leq i \wedge i \leq \max(\text{range\_char}))$

Installing automatic rewrites from: extend
Repeatedly Skolemizing and flattening,
Using lemma char_is_uchar_or_schar,
Splitting conjunctions,
we get 2 subgoals:
`char_binary.1:`

{-1}	$\text{range\_char} = \text{range\_schar}$
{-2}	$\text{integer\_pred}(i')$
{1}	$\text{range\_char}(i') \equiv (\min(\text{range\_char}) \leq i' \wedge i' \leq \max(\text{range\_char}))$

Replacing using formula -1,
Using lemma schar_binary,
This completes the proof of `char_binary.1`.

`char_binary.2:`

{-1}	range_char = extend[int, nat, bool, FALSE](range_uchar)	
{-2}	integer_pred(i')	
{1}	range_char(i') ≡ (min(range_char) ≤ i' ∧ i' ≤ max(range_char))	

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`char_binary.2.1:`

{-1}	range_char = extend[int, nat, bool, FALSE](range_uchar)	
{-2}	integer_pred(i')	
{-3}	extend[int, nat, bool, FALSE](range_uchar)(i')	
{1}	i' ≤ max(extend[int, nat, bool, FALSE](range_uchar))	

Adding type constraints for max(extend[int, nat, bool, FALSE](range_uchar)),

Instantiating the top quantifier in -2 with the terms: (i!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `char_binary.2.1`.

`char_binary.2.2:`

{-1}	range_char = extend[int, nat, bool, FALSE](range_uchar)	
{-2}	integer_pred(i')	
{-3}	extend[int, nat, bool, FALSE](range_uchar)(i')	
{1}	min(extend[int, nat, bool, FALSE](range_uchar)) ≤ i'	

Adding type constraints for min(extend[int, nat, bool, FALSE](range_uchar)),

Instantiating the top quantifier in -2 with the terms: (i!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `char_binary.2.2`.

`char_binary.2.3:`

{-1}	range_char = extend[int, nat, bool, FALSE](range_uchar)	
{-2}	integer_pred(i')	
{-3}	min(extend[int, nat, bool, FALSE](range_uchar)) ≤ i'	
{-4}	i' ≤ max(extend[int, nat, bool, FALSE](range_uchar))	
{1}	extend[int, nat, bool, FALSE](range_uchar)(i')	

Adding type constraints for max(extend[int, nat, bool, FALSE](range_uchar)),

Adding type constraints for min(extend[int, nat, bool, FALSE](range_uchar)),

Using lemma binary_range_uchar,

we get 2 subgoals:

`char_binary.2.3.1:`

```
{-1}  range_uchar(i') ≡ i' ≤ max(range_uchar)
{-2}  extend[int, nat, bool, FALSE]
        (range_uchar)(min(extend[int, nat, bool, FALSE](range_uchar)))
{-3}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          min(extend[int, nat, bool, FALSE](range_uchar)) ≤ x
{-4}  extend[int, nat, bool, FALSE]
        (range_uchar)(max(extend[int, nat, bool, FALSE](range_uchar)))
{-5}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          x ≤ max(extend[int, nat, bool, FALSE](range_uchar))
{-6}  range_char = extend[int, nat, bool, FALSE](range_uchar)
{-7}  integer_pred(i')
{-8}  min(extend[int, nat, bool, FALSE](range_uchar)) ≤ i'
{-9}  i' ≤ max(extend[int, nat, bool, FALSE](range_uchar))
-------------------------------------------------------------
 {1}  extend[int, nat, bool, FALSE](range_uchar)(i')
```

Using lemma binary_range_uchar,

we get 2 subgoals:

`char_binary.2.3.1.1:`

```
{-1}  range_uchar(max(extend[int, nat, bool, FALSE](range_uchar))) ≡
        max(extend[int, nat, bool, FALSE](range_uchar)) ≤ max(range_uchar)
{-2}  range_uchar(i') ≡ i' ≤ max(range_uchar)
{-3}  extend[int, nat, bool, FALSE]
        (range_uchar)(min(extend[int, nat, bool, FALSE](range_uchar)))
{-4}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          min(extend[int, nat, bool, FALSE](range_uchar)) ≤ x
{-5}  extend[int, nat, bool, FALSE]
        (range_uchar)(max(extend[int, nat, bool, FALSE](range_uchar)))
{-6}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          x ≤ max(extend[int, nat, bool, FALSE](range_uchar))
{-7}  range_char = extend[int, nat, bool, FALSE](range_uchar)
{-8}  integer_pred(i')
{-9}  min(extend[int, nat, bool, FALSE](range_uchar)) ≤ i'
{-10} i' ≤ max(extend[int, nat, bool, FALSE](range_uchar))
-------------------------------------------------------------
 {1}  extend[int, nat, bool, FALSE](range_uchar)(i')
```

Expanding the definition of extend,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `char_binary.2.3.1.1`.

`char_binary.2.3.1.2`:

```
{-1}  range_uchar(i') ≡ i' ≤ max(range_uchar)
{-2}  extend[int, nat, bool, FALSE]
        (range_uchar)(min(extend[int, nat, bool, FALSE](range_uchar)))
{-3}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          min(extend[int, nat, bool, FALSE](range_uchar)) ≤ x
{-4}  extend[int, nat, bool, FALSE]
        (range_uchar)(max(extend[int, nat, bool, FALSE](range_uchar)))
{-5}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          x ≤ max(extend[int, nat, bool, FALSE](range_uchar))
{-6}  range_char = extend[int, nat, bool, FALSE](range_uchar)
{-7}  integer_pred(i')
{-8}  min(extend[int, nat, bool, FALSE](range_uchar)) ≤ i'
{-9}  i' ≤ max(extend[int, nat, bool, FALSE](range_uchar))
-------------------------------------------------------------
{1}   max[int, restrict[[real, real], [int, int], boolean[(≤)]]
        (extend[int, nat, bool, FALSE](range_uchar))
        ≥ 0
{2}   extend[int, nat, bool, FALSE](range_uchar)(i')
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `char_binary.2.3.1.2`.
`char_binary.2.3.2`:

```
{-1}  extend[int, nat, bool, FALSE]
        (range_uchar)(min(extend[int, nat, bool, FALSE](range_uchar)))
{-2}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          min(extend[int, nat, bool, FALSE](range_uchar)) ≤ x
{-3}  extend[int, nat, bool, FALSE]
        (range_uchar)(max(extend[int, nat, bool, FALSE](range_uchar)))
{-4}  ∀ (x: int):
        extend[int, nat, bool, FALSE](range_uchar)(x) ⊃
          x ≤ max(extend[int, nat, bool, FALSE](range_uchar))
{-5}  range_char = extend[int, nat, bool, FALSE](range_uchar)
{-6}  integer_pred(i')
{-7}  min(extend[int, nat, bool, FALSE](range_uchar)) ≤ i'
{-8}  i' ≤ max(extend[int, nat, bool, FALSE](range_uchar))
-------------------------------------------------------------
{1}   i' ≥ 0
{2}   extend[int, nat, bool, FALSE](range_uchar)(i')
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `char_binary.2.3.2`.
Q.E.D.

C.50 Proofs for Cpp_class (types.pvs)

C.50.1 Cpp_class.uidt_class_TCC1

Terse proof for `uidt_class_TCC1`.

uidt_class_TCC1:

{1} ∃ (x: [Cpp_Subtype(class?) → {uidt: (uninterpreted_data_type?) | size(uidt) > 0}]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (c: Cpp_Subtype(class?)): choose({uidt: (uninterpreted_data_type?) | size(uidt) > 0})),
we get 2 subgoals:
uidt_class_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of uidt_class_TCC1.1.
uidt_class_TCC1.2:

{1} ∀ (c: Cpp_Subtype(class?)):
nonempty?[(uninterpreted_data_type?)]
({uidt: (uninterpreted_data_type?) | size(uidt) > 0})

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in -3 with the terms: ((# size := 1, valid? := LAMBDA (l: list[Byte], a: Address): FALSE #)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of uidt_class_TCC1.2.
Q.E.D.

C.51 Proofs for Cpp_const_volatile (types.pvs)

C.51.1 Cpp_const_volatile.dt_volatile_TCC1

Terse proof for dt_volatile_TCC1.
dt_volatile_TCC1:

{1} ∃ (x: [(interpreted_data_type?[Data]) → (interpreted_data_type?[Data])]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (idt: (interpreted_data_type?[Data])): idt),
which is trivially true.
This completes the proof of dt_volatile_TCC1.
Q.E.D.

C.52 Proofs for Cpp_double (types.pvs)

C.52.1 Cpp_double.dt_double_TCC1

Terse proof for dt_double_TCC1.
dt_double_TCC1:

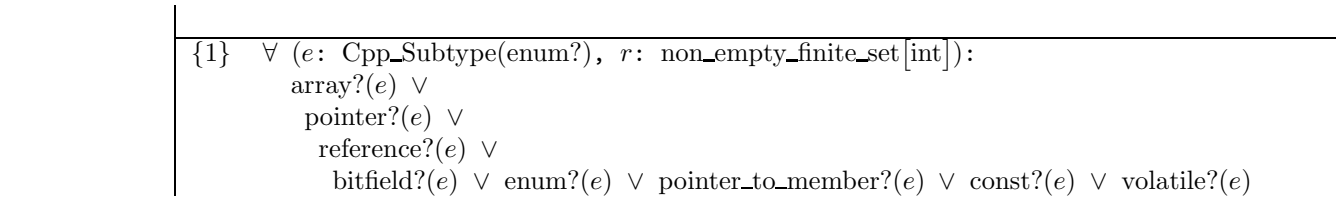
{1} ∃ (x_1: (pod_data_type?[(range_double)])): TRUE

Rewriting using dt_double_exists, matching in *,
This completes the proof of `dt_double_TCC1`.
Q.E.D.

C.53 Proofs for Cpp_enum (types.pvs)

C.53.1 Cpp_enum.valid_range?_TCC1

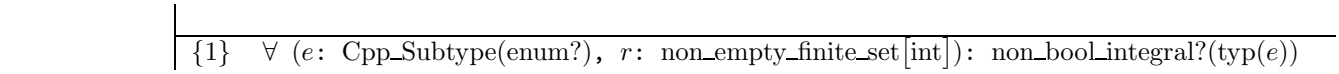
Terse proof for `valid_range?_TCC1`.
`valid_range?_TCC1:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_range?_TCC1`.
Q.E.D.

C.53.2 Cpp_enum.valid_range?_TCC2

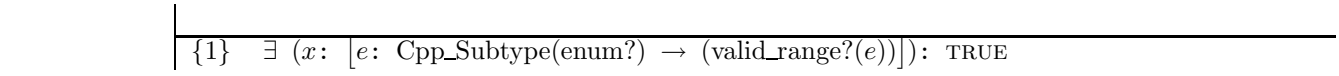
Terse proof for `valid_range?_TCC2`.
`valid_range?_TCC2:`



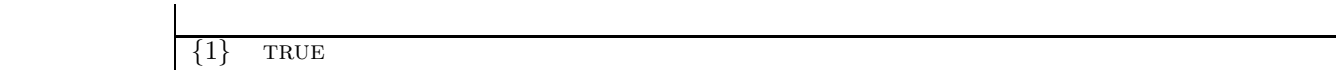
Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Expanding the definition of subterm,
Expanding the definition of enum_underlying_integral?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `valid_range?_TCC2`.
Q.E.D.

C.53.3 Cpp_enum.range_enum_TCC1

Terse proof for `range_enum_TCC1`.
`range_enum_TCC1:`



Instantiating the top quantifier in 1 with the terms: (λ (e: Cpp_Subtype(enum?)): range_integral(typ(e))),
we get 3 subgoals:
`range_enum_TCC1.1:`



which is trivially true.
This completes the proof of `range_enum_TCC1.1`.
`range_enum_TCC1.2`:

{1} ∀ (e: Cpp_Subtype(enum?)): valid_range?(e)(range_integral(typ(e)))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Keeping (-18 -19 1) and hiding *,
Expanding the definition of enum_underlying_integral?,
Expanding the definition of enum_constants?,
Expanding the definition of valid_range?,
Rewriting using subset_reflexive, matching in *,
This completes the proof of `range_enum_TCC1.2`.
`range_enum_TCC1.3`:

{1} ∀ (e: Cpp_Subtype(enum?)): non_bool_integral?(typ(e))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Keeping (-18 1) and hiding *,
Expanding the definition of enum_underlying_integral?,
which is trivially true.
This completes the proof of `range_enum_TCC1.3`.
Q.E.D.

C.53.4 Cpp_enum.dt_enum_TCC1

Terse proof for `dt_enum_TCC1`.
`dt_enum_TCC1`:

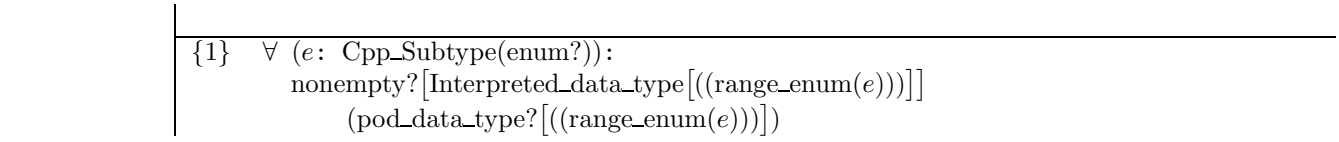
{1} ∃ (x_1: [e: Cpp_Subtype(enum?) → (pod_data_type?[((range_enum(e)))])]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (e: Cpp_Subtype(enum?)): choose(pod_data_type?[((range_enum(e)))])),
we get 2 subgoals:
`dt_enum_TCC1.1`:

{1} TRUE

which is trivially true.
This completes the proof of `dt_enum_TCC1.1`.

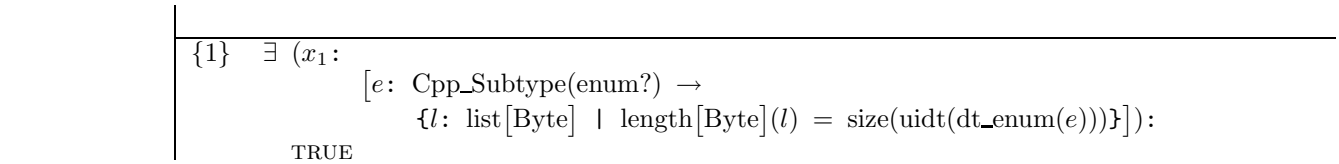
dt_enum_TCC1.2:



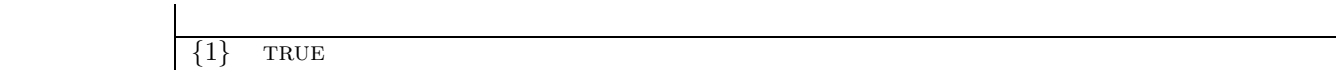
Repeatedly Skolemizing and flattening,
Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
Using lemma dt_enum_exists,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of dt_enum_TCC1.2.
Q.E.D.

C.53.5 Cpp_enum.value_bitmask_enum_TCC1

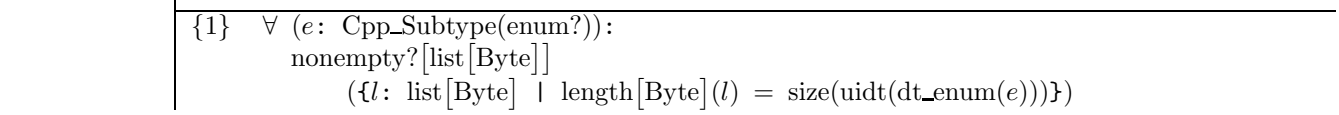
Terse proof for value_bitmask_enum_TCC1.
value_bitmask_enum_TCC1:



Instantiating the top quantifier in 1 with the terms: (λ (e: Cpp_Subtype(enum?)): choose({l: list[Byte] | length[Byte](l) = size(uidt(dt_enum(e)))})),
we get 2 subgoals:
value_bitmask_enum_TCC1.1:



which is trivially true.
This completes the proof of value_bitmask_enum_TCC1.1.
value_bitmask_enum_TCC1.2:



Repeatedly Skolemizing and flattening,
Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
Using lemma list_all_length[Byte],
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -5 with the terms: (l!1),
which is trivially true.
This completes the proof of value_bitmask_enum_TCC1.2.
Q.E.D.

C.53.6 Cpp_enum.enum_size_TCC1

Terse proof for `enum_size_TCC1`.

`enum_size_TCC1:`

{1} ∀ (e: Cpp_Subtype(enum?)):
array?(e) ∨
pointer?(e) ∨
reference?(e) ∨
bitfield?(e) ∨ enum?(e) ∨ pointer_to_member?(e) ∨ const?(e) ∨ volatile?(e)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `enum_size_TCC1`.
Q.E.D.

C.53.7 Cpp_enum.enum_size_TCC2

Terse proof for `enum_size_TCC2`.

`enum_size_TCC2:`

{1} ∀ (e: Cpp_Subtype(enum?)): non_bool_integral?(typ(e))

Repeatedly Skolemizing and flattening,
Expanding the definition of Cpp_Type?,
Instantiating quantified variables,
Expanding the definition of subterm,
Applying disjunctive simplification to flatten sequent,
Keeping (-18 -27 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `enum_size_TCC2`.
Q.E.D.

C.54 Proofs for Cpp_float (types.pvs)

C.54.1 Cpp_float.range_float_TCC1

Terse proof for `range_float_TCC1`.

`range_float_TCC1:`

{1} ∃ (x: non_empty_finite_set[extended_real]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (n: extended_real): $n = 0$),
we get 2 subgoals:

`range_float_TCC1.1:`

{1} TRUE

which is trivially true.
This completes the proof of `range_float_TCC1.1`.

`range_float_TCC1.2`:

{1} is_finite[extended_real](λ (n: extended_real): n = 0) ∧
¬ empty?[extended_real](λ (n: extended_real): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: extended_real): n = 0)): 0),
which is trivially true.
This completes the proof of `range_float_TCC1.2`.
Q.E.D.

C.54.2 Cpp_float.dt_float_TCC1

Terse proof for `dt_float_TCC1`.
`dt_float_TCC1`:

{1} ∃ (x_1: (pod_data_type?[(range_float)])): TRUE

Rewriting using dt_float_exists, matching in *,
This completes the proof of `dt_float_TCC1`.
Q.E.D.

C.55 Proofs for Cpp_function (types.pvs)

C.55.1 Cpp_function.uidt_function_TCC1

Terse proof for `uidt_function_TCC1`.
`uidt_function_TCC1`:

{1} ∀ (f: Cpp_Subtype(function?)):
uninterpreted_data_type?((#size := 0, valid? := λ (l: list[Byte], a: Address): FALSE#))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_function_TCC1`.
Q.E.D.

C.55.2 Cpp_function.dt_vtable_TCC1

Terse proof for `dt_vtable_TCC1`.
`dt_vtable_TCC1`:

{1} ∃ (x_1:
[c: Cpp_Subtype(class?) →
(interpreted_data_type?[((singleton[Vtable](vtable(c))))])]):
TRUE

Instantiating the top quantifier in 1 with the terms: (λ (c: Cpp_Subtype(class?)): choose(interpreted_data_type?[((singleton[Vtable](vtable(c))))])),
we get 2 subgoals:

dt_vtable_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of dt_vtable_TCC1.1.
dt_vtable_TCC1.2:

{1} ∀ (c: Cpp_Subtype(class?)):
nonempty?[Interpreted_data_type[((singleton[Vtable](vtable(c))))]]
(interpreted_data_type?[((singleton[Vtable](vtable(c))))])

Repeatedly Skolemizing and flattening,
Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
Using lemma vtable_idt_exists,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of dt_vtable_TCC1.2.
Q.E.D.

C.56 Proofs for Cpp_int (types.pvs)

C.56.1 Cpp_int.range_int_TCC1

Terse proof for range_int_TCC1.
range_int_TCC1:

{1} ∃ (x: non_empty_finite_set[int]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (n: int): n = 0),
we get 2 subgoals:
range_int_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of range_int_TCC1.1.
range_int_TCC1.2:

{1} is_finite[int](λ (n: int): n = 0) ∧ ¬ empty?[int](λ (n: int): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),
which is trivially true.
This completes the proof of range_int_TCC1.2.
Q.E.D.

C.56.2 Cpp_int.dt_int_TCC1

Terse proof for `dt_int_TCC1`.
`dt_int_TCC1:`

|
|---
| {1} ∃ (x_1: (pod_data_type?[(range_int)])): TRUE

Rewriting using dt_int_exists, matching in *,
This completes the proof of `dt_int_TCC1`.
Q.E.D.

C.56.3 Cpp_int.value_bitmask_int_TCC1

Terse proof for `value_bitmask_int_TCC1`.
`value_bitmask_int_TCC1:`

|
|---
| {1} ∃ (x_1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_int))}): TRUE

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_int_TCC1`.
Q.E.D.

C.57 Proofs for Cpp_long (types.pvs)

C.57.1 Cpp_long.range_long_TCC1

Terse proof for `range_long_TCC1`.
`range_long_TCC1:`

|
|---
| {1} ∃ (x: non_empty_finite_set[int]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (n: int): $n = 0$),
we get 2 subgoals:
`range_long_TCC1.1:`

|
|---
| {1} TRUE

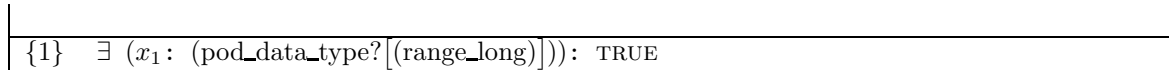
which is trivially true.
This completes the proof of `range_long_TCC1.1`.
`range_long_TCC1.2:`

|
|---
| {1} is_finite[int](λ (n: int): $n = 0$) $\wedge$ $\neg$ empty?[int](λ (n: int): $n = 0$)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),
which is trivially true.
This completes the proof of `range_long_TCC1.2`.
Q.E.D.

C.57.2 Cpp_long.dt_long_TCC1

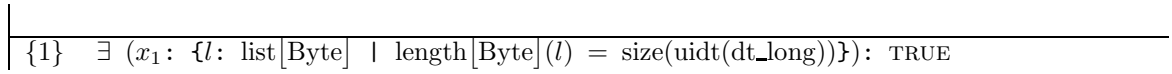
Terse proof for `dt_long_TCC1`.
`dt_long_TCC1`:



Rewriting using dt_long_exists, matching in *,
This completes the proof of `dt_long_TCC1`.
Q.E.D.

C.57.3 Cpp_long.value_bitmask_long_TCC1

Terse proof for `value_bitmask_long_TCC1`.
`value_bitmask_long_TCC1`:

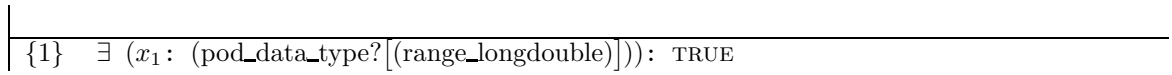


Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_long_TCC1`.
Q.E.D.

C.58 Proofs for Cpp_longdouble (types.pvs)

C.58.1 Cpp_longdouble.dt_longdouble_TCC1

Terse proof for `dt_longdouble_TCC1`.
`dt_longdouble_TCC1`:

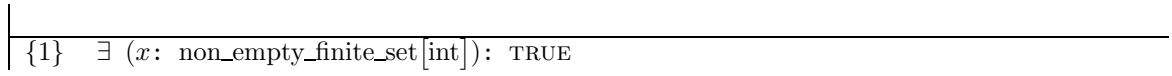


Rewriting using dt_longdouble_exists, matching in *,
This completes the proof of `dt_longdouble_TCC1`.
Q.E.D.

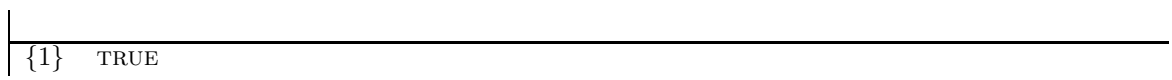
C.59 Proofs for Cpp_longlong (types.pvs)

C.59.1 Cpp_longlong.range_longlong_TCC1

Terse proof for `range_longlong_TCC1`.
`range_longlong_TCC1`:



Instantiating the top quantifier in 1 with the terms: (λ (n: int): n = 0),
we get 2 subgoals:
`range_longlong_TCC1.1`:



which is trivially true.
This completes the proof of `range_longlong_TCC1.1`.
`range_longlong_TCC1.2`:

{1} is_finite[int](λ (n: int): n = 0) ∧ ¬ empty?[int](λ (n: int): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),
which is trivially true.
This completes the proof of `range_longlong_TCC1.2`.
Q.E.D.

C.59.2 Cpp_longlong.dt_longlong_TCC1

Terse proof for `dt_longlong_TCC1`.
`dt_longlong_TCC1`:

{1} ∃ (x1: (pod_data_type?[(range_longlong)])): TRUE

Rewriting using dt_longlong_exists, matching in *,
This completes the proof of `dt_longlong_TCC1`.
Q.E.D.

C.59.3 Cpp_longlong.value_bitmask_longlong_TCC1

Terse proof for `value_bitmask_longlong_TCC1`.
`value_bitmask_longlong_TCC1`:

{1} ∃ (x1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_longlong))}): TRUE

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_longlong_TCC1`.
Q.E.D.

C.60 Proofs for Cpp_pointer (types.pvs)

C.60.1 Cpp_pointer.dt_pointer_TCC1

Terse proof for `dt_pointer_TCC1`.
`dt_pointer_TCC1`:

{1} ∃ (x1: [p: Cpp_Subtype(pointer?) → (pod_data_type?[((range_pointer(p)))])]): TRUE

Using lemma dt_pointer_exists,
Instantiating the top quantifier in 1 with the terms: (LAMBDA (p: Cpp_Subtype(pointer?)): choose(pod_data_type?[((range_pointer(p)))])),
we get 2 subgoals:

dt_pointer_TCC1.1:

{-1} ∀ (p: Cpp_Subtype(pointer?)):
∃ (dt: (pod_data_type?[((range_pointer(p)))])): TRUE

{1} TRUE

which is trivially true.
This completes the proof of dt_pointer_TCC1.1.
dt_pointer_TCC1.2:

{-1} ∀ (p: Cpp_Subtype(pointer?)):
∃ (dt: (pod_data_type?[((range_pointer(p)))])): TRUE

{1} ∀ (p: Cpp_Subtype(pointer?)):
nonempty?[Interpreted_data_type[((range_pointer(p)))]]
(pod_data_type?[((range_pointer(p)))])

Repeatedly Skolemizing and flattening,
Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
Instantiating quantified variables,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of dt_pointer_TCC1.2.
Q.E.D.

C.60.2 Cpp_pointer.pointers_to_layout_compatible_range_TCC1

Terse proof for pointers_to_layout_compatible_range_TCC1.
pointers_to_layout_compatible_range_TCC1:

{1} ∀ (p_1, p_2: Cpp_Subtype(pointer?)):
array?(p_1) ∨
pointer?(p_1) ∨
reference?(p_1) ∨
bitfield?(p_1) ∨ enum?(p_1) ∨ pointer_to_member?(p_1) ∨ const?(p_1) ∨ volatile?(p_1)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pointers_to_layout_compatible_range_TCC1.
Q.E.D.

C.60.3 Cpp_pointer.pointers_to_layout_compatible_range_TCC2

Terse proof for pointers_to_layout_compatible_range_TCC2.
pointers_to_layout_compatible_range_TCC2:

{1} ∀ (p_1, p_2: Cpp_Subtype(pointer?)): Cpp_Type?(typ(p_1))

Repeatedly Skolemizing and flattening,
Keeping (-1 1) and hiding *,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pointers_to_layout_compatible_range_TCC2`.
Q.E.D.

C.60.4 Cpp_pointer.pointers_to_layout_compatible_range_TCC3

Terse proof for `pointers_to_layout_compatible_range_TCC3`.
`pointers_to_layout_compatible_range_TCC3`:

{1} ∀ (p_1, p_2: Cpp_Subtype(pointer?)):
array?(p_2) ∨
pointer?(p_2) ∨
reference?(p_2) ∨
bitfield?(p_2) ∨ enum?(p_2) ∨ pointer_to_member?(p_2) ∨ const?(p_2) ∨ volatile?(p_2)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pointers_to_layout_compatible_range_TCC3`.
Q.E.D.

C.60.5 Cpp_pointer.pointers_to_layout_compatible_range_TCC4

Terse proof for `pointers_to_layout_compatible_range_TCC4`.
`pointers_to_layout_compatible_range_TCC4`:

{1} ∀ (p_1, p_2: Cpp_Subtype(pointer?)): Cpp_Type?(typ(p_2))

Repeatedly Skolemizing and flattening,
Keeping (-3 1) and hiding *,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of subterm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pointers_to_layout_compatible_range_TCC4`.
Q.E.D.

C.60.6 Cpp_pointer.pointers_to_layout_compatible_dt_TCC1

Terse proof for `pointers_to_layout_compatible_dt_TCC1`.
`pointers_to_layout_compatible_dt_TCC1`:

{1} ∀ (p_1, p_2: Cpp_Subtype(pointer?)):
layout_compatible?(typ(p_1), typ(p_2)) ⊃
(∀ (x: lift[Pointer_Base_Type]): range_pointer(p_2)(x) ≡ range_pointer(p_1)(x))

Repeatedly Skolemizing and flattening,
Using lemma pointers_to_layout_compatible_range,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pointers_to_layout_compatible_dt_TCC1`.
Q.E.D.

C.60.7 Cpp_pointer.void_pointer_char_pointer_dt_TCC1

Terse proof for `void_pointer_char_pointer_dt_TCC1`.

`void_pointer_char_pointer_dt_TCC1:`

{1} ∀ (p_1, p_2: Cpp_Subtype(pointer?)):
void?(typ(p_1)) ∧ char?(typ(p_2)) ⊃
(∀ (x: lift[Pointer_Base_Type]): range_pointer(p_2)(x) ≡ range_pointer(p_1)(x))

Repeatedly Skolemizing and flattening,
Using lemma void_pointer_char_pointer_range,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `void_pointer_char_pointer_dt_TCC1`.
Q.E.D.

C.60.8 Cpp_pointer.add_TCC1

Terse proof for `add_TCC1`.

`add_TCC1:`

{1} ∃ (x:
[ptyp: Cpp_Subtype(cv(pointer?)) →
[[(range_pointer(cv_base(ptyp))), int] →
(range_pointer(cv_base(ptyp)))]]):
TRUE

Instantiating the top quantifier in 1 with the terms: (λ (p: Cpp_Subtype(cv(pointer?))): λ (y: (range_pointer(cv_base(p))), z: int): y),
which is trivially true.
This completes the proof of `add_TCC1`.
Q.E.D.

C.60.9 Cpp_pointer.add_spec2_TCC1

Terse proof for `add_spec2_TCC1`.

`add_spec2_TCC1:`

{1} ∀ (i: int, ptyp: Cpp_Subtype(cv(pointer?)), ptr_val: (range_pointer(cv_base(ptyp)))):
not_null?(ptyp)(ptr_val) ⊃
up?[Memory_Address](address_of(ptyp)(add(ptyp)(ptr_val, i)))

Repeatedly Skolemizing and flattening,
Using lemma add_spec,
Using lemma address_of_spec,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `add_spec2_TCC1`.
Q.E.D.

C.60.10 Cpp_pointer.add_spec2_TCC2

Terse proof for `add_spec2_TCC2`.

`add_spec2_TCC2:`

{1} ∀ (ptyp: Cpp_Subtype(cv(pointer?)), ptr_val: (range_pointer(cv_base(ptyp)))):
not_null?(ptyp)(ptr_val) ⊃ up?[Memory_Address](address_of(ptyp)(ptr_val))

Repeatedly Skolemizing and flattening,

Using lemma address_of_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `add_spec2_TCC2`.

Q.E.D.

C.60.11 Cpp_pointer.add_simplification

Terse proof for `add_simplification`.

`add_simplification:`

{1} ∀ (ptyp: Cpp_Subtype(cv(pointer?)), ptr_val: (range_pointer(cv_base(ptyp))),
i, j: int):
address_of(ptyp)(add(ptyp)(add(ptyp)(ptr_val, i), j)) =
address_of(ptyp)(add(ptyp)(ptr_val, $i + j$))

Repeatedly Skolemizing and flattening,

Case splitting on not_null?(ptyp!1)(ptr_val!1),

we get 2 subgoals:

`add_simplification.1:`

{-1} not_null?(ptyp′)(ptr_val′)
{-2} Cpp_Type?(ptyp′)
{-3} cv(pointer?)(ptyp′)
{-4} range_pointer(cv_base(ptyp′))(ptr_val′)
{-5} integer_pred(i')
{-6} integer_pred(j')

{1} address_of(ptyp′)(add(ptyp′)(add(ptyp′)(ptr_val′, i'), j')) =
address_of(ptyp′)(add(ptyp′)(ptr_val′, $i' + j'$))

Using lemma add_spec,

Using lemma add_spec,

Using lemma add_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma add_spec2,

Using lemma add_spec2,

Using lemma add_spec2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -5,

Case splitting on down(address_of(ptyp!1)(ptr_val!1)) + i!1 + j!1 = down(address_of(ptyp!1)(ptr_val!1)) + (i!1 + j!1),

we get 2 subgoals:

`add_simplification.1.1:`

```
{-1}  down(address_of(ptyp')(ptr_val')) + i' + j' =
        down(address_of(ptyp')(ptr_val')) + (i' + j')
{-2}  not_null?(ptyp')(ptr_val')
{-3}  down(address_of(ptyp')(add(ptyp')(ptr_val', i' + j'))) =
        down(address_of(ptyp')(ptr_val')) + i' + j'
{-4}  not_null?(ptyp')(add(ptyp')(ptr_val', i'))
{-5}  down(address_of(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j'))) =
        down(address_of(ptyp')(ptr_val')) + i' + j'
{-6}  down(address_of(ptyp')(add(ptyp')(ptr_val', i'))) =
        down(address_of(ptyp')(ptr_val')) + i'
{-7}  not_null?(ptyp')(add(ptyp')(ptr_val', i' + j'))
{-8}  not_null?(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j'))
{-9}  Cpp_Type?(ptyp')
{-10} cv(pointer?)(ptyp')
{-11} range_pointer(cv_base(ptyp'))(ptr_val')
{-12} integer_pred(i')
{-13} integer_pred(j')
------------------------------------------------------------------
 {1}  address_of(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j')) =
        address_of(ptyp')(add(ptyp')(ptr_val', i' + j'))
```

Replacing using formula -1,
Replacing using formula -3,
Keeping (-5 -7 -8 1) and hiding *,
Using lemma address_of_spec,
Using lemma address_of_spec,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3),
Rewriting using lift_up_extensionality, matching in *,
This completes the proof of `add_simplification.1.1`.
`add_simplification.1.2:`

```
{-1}  not_null?(ptyp')(ptr_val')
{-2}  down(address_of(ptyp')(add(ptyp')(ptr_val', i' + j'))) =
        down(address_of(ptyp')(ptr_val')) + i' + j'
{-3}  not_null?(ptyp')(add(ptyp')(ptr_val', i'))
{-4}  down(address_of(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j'))) =
        down(address_of(ptyp')(ptr_val')) + i' + j'
{-5}  down(address_of(ptyp')(add(ptyp')(ptr_val', i'))) =
        down(address_of(ptyp')(ptr_val')) + i'
{-6}  not_null?(ptyp')(add(ptyp')(ptr_val', i' + j'))
{-7}  not_null?(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j'))
{-8}  Cpp_Type?(ptyp')
{-9}  cv(pointer?)(ptyp')
{-10} range_pointer(cv_base(ptyp'))(ptr_val')
{-11} integer_pred(i')
{-12} integer_pred(j')
------------------------------------------------------------------
 {1}  down(address_of(ptyp')(ptr_val')) + i' + j' =
        down(address_of(ptyp')(ptr_val')) + (i' + j')
 {2}  address_of(ptyp')(add(ptyp')(add(ptyp')(ptr_val', i'), j')) =
        address_of(ptyp')(add(ptyp')(ptr_val', i' + j'))
```

Keeping (1) and hiding *,
Expanding the definition of +,

which is trivially true.
This completes the proof of `add_simplification.1.2`.
`add_simplification.2`:

{-1} Cpp_Type?(ptyp′)
{-2} cv(pointer?)(ptyp′)
{-3} range_pointer(cv_base(ptyp′))(ptr_val′)
{-4} integer_pred(i′)
{-5} integer_pred(j′)

{1} not_null?(ptyp′)(ptr_val′)
{2} address_of(ptyp′)(add(ptyp′)(add(ptyp′)(ptr_val′, i′), j′)) =
address_of(ptyp′)(add(ptyp′)(ptr_val′, i′ + j′))

Using lemma add_spec,
Using lemma add_spec,
Using lemma add_spec,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of not_null?,
Using lemma address_of_spec2,
Using lemma address_of_spec2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `add_simplification.2`.
Q.E.D.

C.60.12 Cpp_pointer.void_pointer_other_pointer_value_TCC1

Terse proof for `void_pointer_other_pointer_value_TCC1`.
`void_pointer_other_pointer_value_TCC1`:

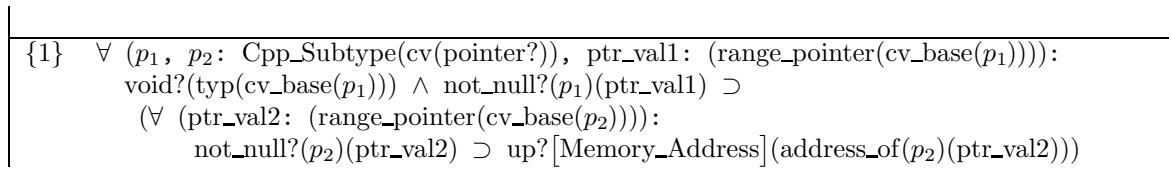
{1} ∀ (p1, p2: Cpp_Subtype(cv(pointer?)), ptr_val1: (range_pointer(cv_base(p1)))):
array?(cv_base(p1)) ∨
pointer?(cv_base(p1)) ∨
reference?(cv_base(p1)) ∨
bitfield?(cv_base(p1)) ∨
enum?(cv_base(p1)) ∨
pointer_to_member?(cv_base(p1)) ∨ const?(cv_base(p1)) ∨ volatile?(cv_base(p1))

Repeatedly Skolemizing and flattening,
Keeping (-2 2) and hiding *,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `void_pointer_other_pointer_value_TCC1`.
Q.E.D.

C.60.13 Cpp_pointer.void_pointer_other_pointer_value_TCC2

Terse proof for `void_pointer_other_pointer_value_TCC2`.

`void_pointer_other_pointer_value_TCC2:`



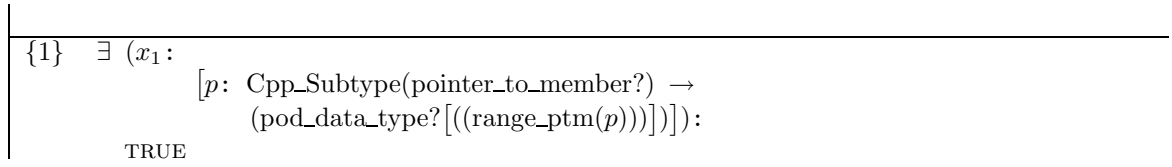
Repeatedly Skolemizing and flattening,
Rewriting using address_of_spec, matching in *,
This completes the proof of `void_pointer_other_pointer_value_TCC2`.
Q.E.D.

C.61 Proofs for Cpp_pointer_to_member (types.pvs)

C.61.1 Cpp_pointer_to_member.dt_ptm_TCC1

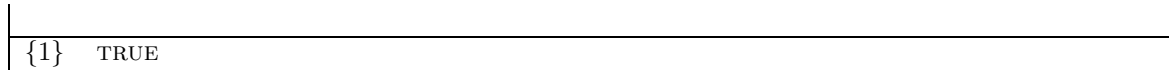
Terse proof for `dt_ptm_TCC1`.

`dt_ptm_TCC1:`



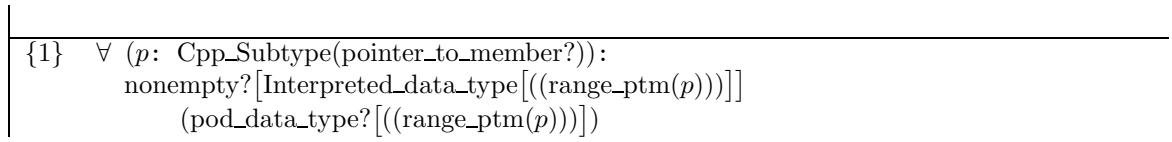
Instantiating the top quantifier in 1 with the terms: (λ (p: Cpp_Subtype(pointer_to_member?)): choose(pod_data_type?[((range_ptm(p)))])),
we get 2 subgoals:

`dt_ptm_TCC1.1:`



which is trivially true.
This completes the proof of `dt_ptm_TCC1.1`.

`dt_ptm_TCC1.2:`



Repeatedly Skolemizing and flattening,
Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
Using lemma dt_pointer_to_member_exists,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `dt_ptm_TCC1.2`.
Q.E.D.

C.62 Proofs for Cpp_reference (types.pvs)

C.62.1 Cpp_reference.dt_reference_TCC1

Terse proof for `dt_reference_TCC1`.

`dt_reference_TCC1:`

{1} ∃ (x_1: ⌊r: Cpp_Subtype(reference?) → (pod_data_type?⌊((alloc_range(r)))⌋)⌋): TRUE

Using lemma dt_ref_exists,

Instantiating the top quantifier in 1 with the terms: (LAMBDA (r: Cpp_Subtype(reference?)): choose(pod_data_type?[((alloc_range(r)))])),

we get 2 subgoals:

`dt_reference_TCC1.1:`

{-1} ∀ (r: Cpp_Subtype(reference?)): ∃ (dt: (pod_data_type?⌊((alloc_range(r)))⌋)): TRUE

{1} TRUE

which is trivially true.

This completes the proof of `dt_reference_TCC1.1`.

`dt_reference_TCC1.2:`

{-1} ∀ (r: Cpp_Subtype(reference?)): ∃ (dt: (pod_data_type?⌊((alloc_range(r)))⌋)): TRUE

{1} ∀ (r: Cpp_Subtype(reference?)): nonempty?⌊Interpreted_data_type⌊((alloc_range(r)))⌋⌋ (pod_data_type?⌊((alloc_range(r)))⌋)

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of member,

which is trivially true.

This completes the proof of `dt_reference_TCC1.2`.

Q.E.D.

C.63 Proofs for Cpp_schar (types.pvs)

C.63.1 Cpp_schar.range_schar_TCC1

Terse proof for `range_schar_TCC1`.

`range_schar_TCC1:`

{1} ∃ (x: non_empty_finite_set⌊int⌋): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (n: int): $n = 0$),

we get 2 subgoals:

range_schar_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of range_schar_TCC1.1.
range_schar_TCC1.2:

{1} is_finite[int](λ (n: int): n = 0) ∧ ¬ empty?[int](λ (n: int): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),
which is trivially true.
This completes the proof of range_schar_TCC1.2.
Q.E.D.

C.63.2 Cpp_schar.dt_schar_TCC1

Terse proof for dt_schar_TCC1.
dt_schar_TCC1:

{1} ∃ (x1: (pod_data_type?[(range_schar)])): TRUE

Rewriting using dt_schar_exists, matching in *,
This completes the proof of dt_schar_TCC1.
Q.E.D.

C.63.3 Cpp_schar.schar_value_eq_uchar_value_TCC1

Terse proof for schar_value_eq_uchar_value_TCC1.
schar_value_eq_uchar_value_TCC1:

{1} ∀ (i: (range_schar)): i ≥ 0 ⊃ (range_uchar)(i)

Repeatedly Skolemizing and flattening,
Rewriting using schar_nonneg_is_uchar, matching in *,
This completes the proof of schar_value_eq_uchar_value_TCC1.
Q.E.D.

C.63.4 Cpp_schar.value_bitmask_schar_TCC1

Terse proof for value_bitmask_schar_TCC1.
value_bitmask_schar_TCC1:

{1} ∃ (x1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_schar))}): TRUE

Using lemma list_all_length[Byte],
Instantiating the top quantifier in -1 with the terms: (size(uidt(dt_schar))),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_schar_TCC1.

Q.E.D.

C.64 Proofs for Cpp_short (types.pvs)

C.64.1 Cpp_short.range_short_TCC1

Terse proof for `range_short_TCC1`.

`range_short_TCC1:`

|
|---|
| {1} ∃ (x: non_empty_finite_set[int]): TRUE |

Instantiating the top quantifier in 1 with the terms: (λ (n: int): $n = 0$),
we get 2 subgoals:

`range_short_TCC1.1:`

|
|---|
| {1} TRUE |

which is trivially true.
This completes the proof of `range_short_TCC1.1`.

`range_short_TCC1.2:`

|
|---|
| {1} is_finite[int](λ (n: int): $n = 0$) $\wedge$ $\neg$ empty?[int](λ (n: int): $n = 0$) |

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),
which is trivially true.
This completes the proof of `range_short_TCC1.2`.
Q.E.D.

C.64.2 Cpp_short.dt_short_TCC1

Terse proof for `dt_short_TCC1`.

`dt_short_TCC1:`

|
|---|
| {1} ∃ (x_1: (pod_data_type?[(range_short)])): TRUE |

Rewriting using dt_short_exists, matching in *,
This completes the proof of `dt_short_TCC1`.
Q.E.D.

C.64.3 Cpp_short.value_bitmask_short_TCC1

Terse proof for `value_bitmask_short_TCC1`.

`value_bitmask_short_TCC1:`

|
|---|
| {1} ∃ (x_1: {l: list[Byte] \| length[Byte](l) = size(uidt(dt_short))}): TRUE |

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of value_bitmask_short_TCC1.
Q.E.D.

C.65 Proofs for Cpp_uchar (types.pvs)

C.65.1 Cpp_uchar.range_uchar_TCC1

Terse proof for range_uchar_TCC1.
range_uchar_TCC1:

|
|---|
| {1} ∃ (x: non_empty_finite_set[nat]): TRUE |

Instantiating the top quantifier in 1 with the terms: (λ (n: nat): n = 0),
we get 2 subgoals:
range_uchar_TCC1.1:

|
|---|
| {1} TRUE |

which is trivially true.
This completes the proof of range_uchar_TCC1.1.
range_uchar_TCC1.2:

|
|---|
| {1} is_finite[nat](λ (n: nat): n = 0) ∧ ¬ empty?[nat](λ (n: nat): n = 0) |

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: nat): n = 0)): 0),
which is trivially true.
This completes the proof of range_uchar_TCC1.2.
Q.E.D.

C.65.2 Cpp_uchar.dt_uchar_TCC1

Terse proof for dt_uchar_TCC1.
dt_uchar_TCC1:

|
|---|
| {1} ∃ (x1: (pod_data_type?[(range_uchar)])): TRUE |

Using lemma dt_uchar_exists,
This completes the proof of dt_uchar_TCC1.
Q.E.D.

C.65.3 Cpp_uchar.value_bitmask_uchar_TCC1

Terse proof for value_bitmask_uchar_TCC1.
value_bitmask_uchar_TCC1:

|
|---|
| {1} ∃ (x1: {l: list[Byte] \| length[Byte](l) = size(uidt(dt_uchar))}): TRUE |

Using lemma list_all_length[Byte],

Instantiating the top quantifier in -1 with the terms: (size(uidt(dt_uchar))),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_uchar_TCC1`.
Q.E.D.

C.66 Proofs for Cpp_uint (types.pvs)

C.66.1 Cpp_uint.range_uint_TCC1

Terse proof for `range_uint_TCC1`.
`range_uint_TCC1:`

{1} $\exists$ (x: non_empty_finite_set[nat]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (n: nat): $n = 0$),
we get 2 subgoals:
`range_uint_TCC1.1:`

{1} TRUE

which is trivially true.
This completes the proof of `range_uint_TCC1.1`.
`range_uint_TCC1.2:`

{1} is_finite[nat](λ (n: nat): $n = 0$) $\wedge$ $\neg$ empty?[nat](λ (n: nat): $n = 0$)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (p: (LAMBDA (n: nat): n = 0)): 0),
which is trivially true.
This completes the proof of `range_uint_TCC1.2`.
Q.E.D.

C.66.2 Cpp_uint.dt_uint_TCC1

Terse proof for `dt_uint_TCC1`.
`dt_uint_TCC1:`

{1} $\exists$ (x_1: (pod_data_type?[(range_uint)])): TRUE

Rewriting using dt_uint_exists, matching in *,
This completes the proof of `dt_uint_TCC1`.
Q.E.D.

C.66.3 Cpp_uint.uint_represenation_same_for_positive_int_TCC1

Terse proof for `uint_represenation_same_for_positive_int_TCC1`.
`uint_represenation_same_for_positive_int_TCC1:`

{1} $\forall$ (i: (range_int)): $i \geq 0 \supset$ (range_uint)(i)

Repeatedly Skolemizing and flattening,
Rewriting using uint_value_supset_int_value, matching in *,
This completes the proof of `uint_represenation_same_for_positive_int_TCC1`.
Q.E.D.

C.66.4 Cpp_uint.value_bitmask_uint_TCC1

Terse proof for `value_bitmask_uint_TCC1`.
`value_bitmask_uint_TCC1`:

$$\{1\}\quad \exists\ (x_1\colon \{l\colon \mathrm{list}[\mathrm{Byte}]\ \mid\ \mathrm{length}[\mathrm{Byte}](l) = \mathrm{size}(\mathrm{uidt}(\mathrm{dt\_uint}))\})\colon\ \textsc{true}$$

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_uint_TCC1`.
Q.E.D.

C.67 Proofs for Cpp_ulong (types.pvs)

C.67.1 Cpp_ulong.range_ulong_TCC1

Terse proof for `range_ulong_TCC1`.
`range_ulong_TCC1`:

$$\{1\}\quad \exists\ (x\colon \mathrm{non\_empty\_finite\_set}[\mathrm{nat}])\colon\ \textsc{true}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda\ (n\colon \mathrm{nat})\colon n = 0)$,
we get 2 subgoals:
`range_ulong_TCC1.1`:

$$\{1\}\quad \textsc{true}$$

which is trivially true.
This completes the proof of `range_ulong_TCC1.1`.
`range_ulong_TCC1.2`:

$$\{1\}\quad \mathrm{is\_finite}[\mathrm{nat}](\lambda\ (n\colon \mathrm{nat})\colon n = 0) \wedge \neg\ \mathrm{empty?}[\mathrm{nat}](\lambda\ (n\colon \mathrm{nat})\colon n = 0)$$

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: nat): n = 0)): 0),
which is trivially true.
This completes the proof of `range_ulong_TCC1.2`.
Q.E.D.

C.67.2 Cpp_ulong.dt_ulong_TCC1

Terse proof for `dt_ulong_TCC1`.

dt_ulong_TCC1:

{1} ∃ (x_1: (pod_data_type?[(range_ulong)])): TRUE

Rewriting using dt_ulong_exists, matching in *,
This completes the proof of dt_ulong_TCC1.
Q.E.D.

C.67.3 Cpp_ulong.ulong_represenation_same_for_positive_long_TCC1

Terse proof for ulong_represenation_same_for_positive_long_TCC1.
ulong_represenation_same_for_positive_long_TCC1:

{1} ∀ (i: (range_long)): $i \geq 0 \supset$ (range_ulong)(i)

Repeatedly Skolemizing and flattening,
Rewriting using ulong_value_supset_long_value, matching in *,
This completes the proof of ulong_represenation_same_for_positive_long_TCC1.
Q.E.D.

C.67.4 Cpp_ulong.value_bitmask_ulong_TCC1

Terse proof for value_bitmask_ulong_TCC1.
value_bitmask_ulong_TCC1:

{1} ∃ (x_1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_ulong))}): TRUE

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_ulong_TCC1.
Q.E.D.

C.68 Proofs for Cpp_ulonglong (types.pvs)

C.68.1 Cpp_ulonglong.range_ulonglong_TCC1

Terse proof for range_ulonglong_TCC1.
range_ulonglong_TCC1:

{1} ∃ (x: non_empty_finite_set[nat]): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (n: nat): $n = 0$),
we get 2 subgoals:
range_ulonglong_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of range_ulonglong_TCC1.1.

range_ulonglong_TCC1.2:

{1} is_finite[nat](λ (n: nat): n = 0) ∧ ¬ empty?[nat](λ (n: nat): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: nat): n = 0)): 0),
which is trivially true.
This completes the proof of range_ulonglong_TCC1.2.
Q.E.D.

C.68.2 Cpp_ulonglong.dt_ulonglong_TCC1

Terse proof for dt_ulonglong_TCC1.
dt_ulonglong_TCC1:

{1} ∃ (x1: (pod_data_type?[(range_ulonglong)])): TRUE

Rewriting using dt_ulonglong_exists, matching in *,
This completes the proof of dt_ulonglong_TCC1.
Q.E.D.

C.68.3 Cpp_ulonglong.ulonglong_represenation_same_for_positive_longlong_TCC1

Terse proof for ulonglong_represenation_same_for_positive_longlong_TCC1.
ulonglong_represenation_same_for_positive_longlong_TCC1:

{1} ∀ (i: (range_longlong)): i ≥ 0 ⊃ (range_ulonglong)(i)

Repeatedly Skolemizing and flattening,
Rewriting using ulonglong_value_supset_longlong_value, matching in *,
This completes the proof of ulonglong_represenation_same_for_positive_longlong_TCC1.
Q.E.D.

C.68.4 Cpp_ulonglong.value_bitmask_ulonglong_TCC1

Terse proof for value_bitmask_ulonglong_TCC1.
value_bitmask_ulonglong_TCC1:

{1} ∃ (x1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_ulonglong))}): TRUE

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_ulonglong_TCC1.
Q.E.D.

C.69 Proofs for Cpp_union (types.pvs)

C.69.1 Cpp_union.uidt_union_TCC1

Terse proof for `uidt_union_TCC1`.

`uidt_union_TCC1:`

```
{1}  ∃ (x:
          [Cpp_Subtype(union?) → {uidt: (uninterpreted_data_type?) | size(uidt) > 0}]):
       TRUE
```

Instantiating the top quantifier in 1 with the terms: (λ (c: Cpp_Subtype(union?)): choose({uidt: (uninterpreted_data_type?) | size(uidt) > 0})),

we get 2 subgoals:

`uidt_union_TCC1.1:`

```
{1}  TRUE
```

which is trivially true.

This completes the proof of `uidt_union_TCC1.1`.

`uidt_union_TCC1.2:`

```
{1}  ∀ (c: Cpp_Subtype(union?)):
       nonempty?[(uninterpreted_data_type?)]
            ({uidt: (uninterpreted_data_type?) | size(uidt) > 0})
```

Trying repeated skolemization, instantiation, and if-lifting,

Instantiating the top quantifier in -3 with the terms: ((# size := 1, valid? := LAMBDA (l: list[Byte], a: Address): FALSE #)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `uidt_union_TCC1.2`.

Q.E.D.

C.70 Proofs for Cpp_ushort (types.pvs)

C.70.1 Cpp_ushort.range_ushort_TCC1

Terse proof for `range_ushort_TCC1`.

`range_ushort_TCC1:`

```
{1}  ∃ (x: non_empty_finite_set[nat]): TRUE
```

Instantiating the top quantifier in 1 with the terms: (λ (n: nat): n = 0),

we get 2 subgoals:

`range_ushort_TCC1.1:`

```
{1}  TRUE
```

which is trivially true.

This completes the proof of `range_ushort_TCC1.1`.

range_ushort_TCC1.2:

{1} is_finite[nat](λ (n: nat): n = 0) ∧ ¬ empty?[nat](λ (n: nat): n = 0)

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: nat): n = 0)): 0),
which is trivially true.
This completes the proof of range_ushort_TCC1.2.
Q.E.D.

C.70.2 Cpp_ushort.dt_ushort_TCC1

Terse proof for dt_ushort_TCC1.
dt_ushort_TCC1:

{1} ∃ (x1: (pod_data_type?[(range_ushort)])): TRUE

Rewriting using dt_ushort_exists, matching in *,
This completes the proof of dt_ushort_TCC1.
Q.E.D.

C.70.3 Cpp_ushort.ushort_represenation_same_for_positive_short_TCC1

Terse proof for ushort_represenation_same_for_positive_short_TCC1.
ushort_represenation_same_for_positive_short_TCC1:

{1} ∀ (i: (range_short)): i ≥ 0 ⊃ (range_ushort)(i)

Repeatedly Skolemizing and flattening,
Rewriting using ushort_value_supset_short_value, matching in *,
This completes the proof of ushort_represenation_same_for_positive_short_TCC1.
Q.E.D.

C.70.4 Cpp_ushort.value_bitmask_ushort_TCC1

Terse proof for value_bitmask_ushort_TCC1.
value_bitmask_ushort_TCC1:

{1} ∃ (x1: {l: list[Byte] | length[Byte](l) = size(uidt(dt_ushort))}): TRUE

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of value_bitmask_ushort_TCC1.
Q.E.D.

C.71 Proofs for Cpp_wchar_t (types.pvs)

C.71.1 Cpp_wchar_t.range_wchar_t_TCC1

Terse proof for `range_wchar_t_TCC1`.

`range_wchar_t_TCC1:`

$$\{1\} \quad \exists\ (x\colon \text{non\_empty\_finite\_set}[\text{int}])\colon \text{TRUE}$$

Instantiating the top quantifier in 1 with the terms: $(\lambda\ (n\colon \text{int})\colon n = 0)$,
we get 2 subgoals:

`range_wchar_t_TCC1.1:`

$$\{1\} \quad \text{TRUE}$$

which is trivially true.
This completes the proof of `range_wchar_t_TCC1.1`.

`range_wchar_t_TCC1.2:`

$$\{1\} \quad \text{is\_finite}[\text{int}](\lambda\ (n\colon \text{int})\colon n = 0) \wedge \neg\ \text{empty?}[\text{int}](\lambda\ (n\colon \text{int})\colon n = 0)$$

Trying repeated skolemization, instantiation, and if-lifting,
Instantiating the top quantifier in 1 with the terms: (1 LAMBDA (P: (LAMBDA (n: int): n = 0)): 0),
which is trivially true.
This completes the proof of `range_wchar_t_TCC1.2`.
Q.E.D.

C.71.2 Cpp_wchar_t.dt_wchar_t_TCC1

Terse proof for `dt_wchar_t_TCC1`.

`dt_wchar_t_TCC1:`

$$\{1\} \quad \exists\ (x_1\colon (\text{pod\_data\_type?}[(\text{range\_wchar\_t})]))\colon \text{TRUE}$$

Rewriting using dt_wchar_t_exists, matching in *,
This completes the proof of `dt_wchar_t_TCC1`.
Q.E.D.

C.71.3 Cpp_wchar_t.value_bitmask_wchar_t_TCC1

Terse proof for `value_bitmask_wchar_t_TCC1`.

`value_bitmask_wchar_t_TCC1:`

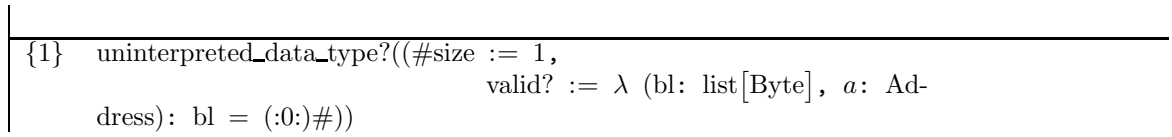
$$\{1\} \quad \exists\ (x_1\colon \{l\colon \text{list}[\text{Byte}]\ \mid\ \text{length}[\text{Byte}](l) = \text{size}(\text{uidt}(\text{dt\_wchar\_t}))\})\colon \text{TRUE}$$

Using lemma list_all_length[Byte],
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_bitmask_wchar_t_TCC1`.
Q.E.D.

C.72 Proofs for Datatype_Model (datatype_model.pvs)

C.72.1 Datatype_Model.uidt_1_TCC1

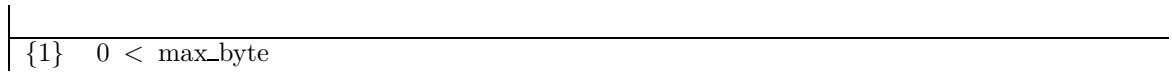
Terse proof for `uidt_1_TCC1`.
`uidt_1_TCC1`:



Trying repeated skolemization, instantiation, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_1_TCC1`.
Q.E.D.

C.72.2 Datatype_Model.pod_1_TCC1

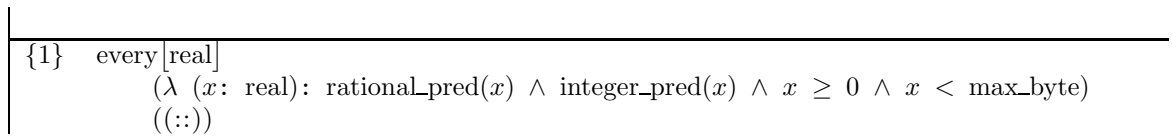
Terse proof for `pod_1_TCC1`.
`pod_1_TCC1`:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_1_TCC1`.
Q.E.D.

C.72.3 Datatype_Model.pod_1_TCC2

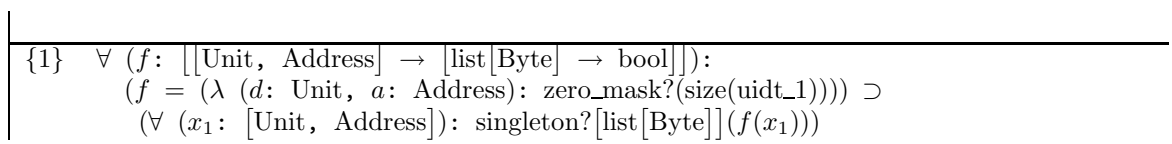
Terse proof for `pod_1_TCC2`.
`pod_1_TCC2`:



Trying repeated skolemization, instantiation, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_1_TCC2`.
Q.E.D.

C.72.4 Datatype_Model.pod_1_TCC3

Terse proof for `pod_1_TCC3`.
`pod_1_TCC3`:



Repeatedly Skolemizing and flattening,
Replacing using formula -1,

Using lemma singleton_zero_mask,
Expanding the definition of singleton?,
which is trivially true.
This completes the proof of `pod_1_TCC3`.
Q.E.D.

C.72.5 Datatype_Model.pod_1_TCC4

Terse proof for `pod_1_TCC4`.
`pod_1_TCC4`:

```
{1}   pod_data_type?[Unit]
          ((#uidt := uidt_1,
                to_byte := λ (d: Unit, a: Address): (:0:),
                to_mask
                  := λ (f: [[Unit, Address] → [list[Byte] → bool]]):
                        λ (x1: [Unit, Address]): singleton_elt[list[Byte]](f(x1))
                          (λ (d: Unit, a: Address): zero_mask?(size(uidt_1))),
                from_byte
                  := λ (bl: list[Byte], a: Address):
                        IF bl = (:0:) THEN up[Unit](unit) ELSE bottom[Unit] ENDIF#))
```

Expanding the definition of pod_data_type?,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_1_TCC4`.
Q.E.D.

C.72.6 Datatype_Model.uidt_2_TCC1

Terse proof for `uidt_2_TCC1`.
`uidt_2_TCC1`:

```
{1}   uninterpreted_data_type?((#size := 1,
                                valid? := λ (bl: list[Byte], a: Ad-
      dress): bl = (:1:)#))
```

Trying repeated skolemization, instantiation, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `uidt_2_TCC1`.
Q.E.D.

C.72.7 Datatype_Model.pod_2_TCC1

Terse proof for `pod_2_TCC1`.
`pod_2_TCC1`:

```
{1}   1 < max_byte
```

Trying repeated skolemization, instantiation, and if-lifting,
Applying less_than_max_byte
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pod_2_TCC1.
Q.E.D.

C.72.8 Datatype_Model.pod_2_TCC2

Terse proof for pod_2_TCC2.

pod_2_TCC2:

{1} ∀ (f: [[Unit, Address] → [list[Byte] → bool]]):
(f = (λ (d: Unit, a: Address): zero_mask?(size(uidt_2)))) ⊃
(∀ (x1: [Unit, Address]): singleton?[list[Byte]](f(x1)))

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Expanding the definition of singleton?,
which is trivially true.
This completes the proof of pod_2_TCC2.
Q.E.D.

C.72.9 Datatype_Model.pod_2_TCC3

Terse proof for pod_2_TCC3.

pod_2_TCC3:

{1} pod_data_type?[Unit]
((#uidt := uidt_2,
to_byte := λ (d: Unit, a: Address): (:1:),
to_mask
:= λ (f: [[Unit, Address] → [list[Byte] → bool]]):
λ (x1: [Unit, Address]): singleton_elt[list[Byte]](f(x1))
(λ (d: Unit, a: Address): zero_mask?(size(uidt_2))),
from_byte
:= λ (bl: list[Byte], a: Address):
IF bl = (:1:) THEN up[Unit](unit) ELSE bottom[Unit] ENDIF#))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pod_2_TCC3.
Q.E.D.

C.73 Proofs for Datatype_Model_2 (datatype_model.pvs)

C.73.1 Datatype_Model_2.uidt_bool_TCC1

Terse proof for uidt_bool_TCC1.

`uidt_bool_TCC1:`

```
{1}  uninterpreted_data_type?((#size := 2,
                               valid?
                                 := λ (bl: list[Byte], a: Address):
                                      bl = (:0, 0:) ∨ bl = (:0, 1:)#))
```

Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:
`uidt_bool_TCC1.1:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧
                   integer_pred(x) ∧ x ≥ 0 ∧ x < expt(2, bits_per_byte))
            ((:0, 0:))
{-2}  l′ = (:0, 0:)
-------------------------------------------------------------------------
 {1}  length((:0, 0:)) = 2
```

Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
which is trivially true.
This completes the proof of `uidt_bool_TCC1.1`.
`uidt_bool_TCC1.2:`

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧
                   integer_pred(x) ∧ x ≥ 0 ∧ x < expt(2, bits_per_byte))
            ((:0, 1:))
{-2}  l′ = (:0, 1:)
-------------------------------------------------------------------------
 {1}  length((:0, 1:)) = 2
```

Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
which is trivially true.
This completes the proof of `uidt_bool_TCC1.2`.
Q.E.D.

C.73.2 Datatype_Model_2.pod_bool_TCC1

Terse proof for `pod_bool_TCC1`.
`pod_bool_TCC1:`

```
{1}  ∀ (d: bool): d ⊃ 0 < max_byte
```

Applying less_than_max_byte
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_bool_TCC1`.
Q.E.D.

C.73.3 Datatype_Model_2.pod_bool_TCC2

Terse proof for `pod_bool_TCC2`.

`pod_bool_TCC2:`

$$\{1\}\quad \forall\ (d\colon \text{bool})\colon\ d\ \supset\ \text{every}[\text{real}]\ (\lambda\ (x\colon \text{real})\colon \text{rational\_pred}(x)\ \wedge\ \text{integer\_pred}(x)\ \wedge\ x \geq 0\ \wedge\ x < \text{max\_byte})\ ((:1:))$$

Trying repeated skolemization, instantiation, and if-lifting,
Applying less_than_max_byte
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_bool_TCC2`.
Q.E.D.

C.73.4 Datatype_Model_2.pod_bool_TCC3

Terse proof for `pod_bool_TCC3`.

`pod_bool_TCC3:`

$$\{1\}\quad \forall\ (d\colon \text{bool})\colon\ \neg\ d\ \supset\ 0 < \text{max\_byte}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_bool_TCC3`.
Q.E.D.

C.73.5 Datatype_Model_2.pod_bool_TCC4

Terse proof for `pod_bool_TCC4`.

`pod_bool_TCC4:`

$$\{1\}\quad \forall\ (d\colon \text{bool})\colon\ \neg\ d\ \supset\ \text{every}[\text{real}]\ (\lambda\ (x\colon \text{real})\colon \text{rational\_pred}(x)\ \wedge\ \text{integer\_pred}(x)\ \wedge\ x \geq 0\ \wedge\ x < \text{max\_byte})\ ((:0:))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_bool_TCC4`.
Q.E.D.

C.73.6 Datatype_Model_2.pod_bool_TCC5

Terse proof for `pod_bool_TCC5`.

`pod_bool_TCC5:`

$$\{1\}\quad \forall\ (f\colon [[\text{bool}, \text{Address}] \rightarrow [\text{list}[\text{Byte}] \rightarrow \text{bool}]])\colon\ (f = (\lambda\ (d\colon \text{bool}, a\colon \text{Address})\colon \text{zero\_mask?}(\text{size}(\text{uidt\_bool})))) \supset\ (\forall\ (x_1\colon [\text{bool}, \text{Address}])\colon \text{singleton?}[\text{list}[\text{Byte}]](f(x_1)))$$

Repeatedly Skolemizing and flattening,

Replacing using formula -1,

Using lemma singleton_zero_mask,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pod_bool_TCC5`.

Q.E.D.

C.73.7 Datatype_Model_2.pod_bool_TCC6

Terse proof for `pod_bool_TCC6`.

`pod_bool_TCC6:`

```
{1}  pod_data_type?[bool]
       ((#uidt := uidt_bool,
           to_byte := λ (d: bool, a: Address): IF d THEN (:0, 1:) ELSE (:0, 0:) ENDIF,
           to_mask
             := λ (f: [[bool, Address] → [list[Byte] → bool]]):
                  λ (x1: [bool, Address]): singleton_elt[list[Byte]](f(x1))
                    (λ (d: bool, a: Address): zero_mask?(size(uidt_bool))),
           from_byte
             := λ (bl: list[Byte], a: Address):
                  IF bl = (:0, 1:)
                    THEN up[boolean](TRUE)
                  ELSE IF bl = (:0, 0:) THEN up[boolean](FALSE) ELSE bottom[boolean] ENDIF
                  ENDIF#))
```

Trying repeated skolemization, instantiation, and if-lifting,

Using lemma min_bits_per_byte,

Using lemma bits_per_byte_minimum,

Using lemma pos_expt_gt,

Case splitting on 1 < expt(2, bits_per_byte),

we get 2 subgoals:

`pod_bool_TCC6.1:`

```
{-1}  1 < expt(2, bits_per_byte)
{-2}  bits_per_byte < 2 ^ bits_per_byte
{-3}  bits_per_byte ≥ min_bits_per_byte
{-4}  min_bits_per_byte = 8
{-5}  (:0, 0:) = (:0, 1:)
-------------------------
 {1}  d'
```

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `pod_bool_TCC6.1`.

pod_bool_TCC6.2:

```
{-1}  bits_per_byte < 2 ^ bits_per_byte
{-2}  bits_per_byte ≥ min_bits_per_byte
{-3}  min_bits_per_byte = 8
{-4}  (:0, 0:) = (:0, 1:)
-------------------------------------------
{1}   1 < expt(2, bits_per_byte)
{2}   d′
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pod_bool_TCC6.2.
Q.E.D.

C.73.8 Datatype_Model_2.uidt_Byte_TCC1

Terse proof for uidt_Byte_TCC1.
uidt_Byte_TCC1:

```
-------------------------------------------
{1}   ∀ (bl: list[Byte]): length(bl) = 2 ⊃ cons?[Byte](bl)
```

Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of length,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_Byte_TCC1.
Q.E.D.

C.73.9 Datatype_Model_2.uidt_Byte_TCC2

Terse proof for uidt_Byte_TCC2.
uidt_Byte_TCC2:

```
-------------------------------------------
{1}   uninterpreted_data_type?((#size := 2,
                                valid?
                                  := λ (bl: list[Byte], a: Address):
                                       length[Byte](bl) = 2 ∧ car[Byte](bl) = 1#))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of uidt_Byte_TCC2.
Q.E.D.

C.73.10 Datatype_Model_2.pod_Byte_TCC1

Terse proof for pod_Byte_TCC1.
pod_Byte_TCC1:

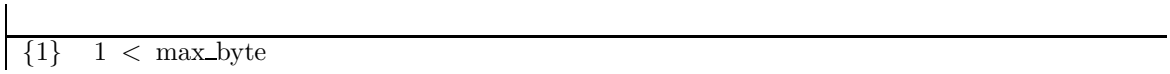
```
-------------------------------------------
{1}   ∀ (bl: list[Byte]): length(bl) = 2 ∧ car(bl) = 1 ⊃ cons?[Byte](cdr[Byte](bl))
```

Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of length,
Expanding the definition of length,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pod_Byte_TCC1.

Q.E.D.

C.73.11 Datatype_Model_2.pod_Byte_TCC2

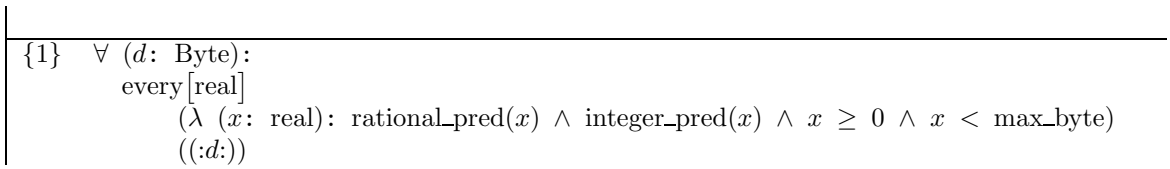
Terse proof for `pod_Byte_TCC2`.
`pod_Byte_TCC2:`



Trying repeated skolemization, instantiation, and if-lifting,
Applying less_than_max_byte
Expanding the definition of max_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_Byte_TCC2`.
Q.E.D.

C.73.12 Datatype_Model_2.pod_Byte_TCC3

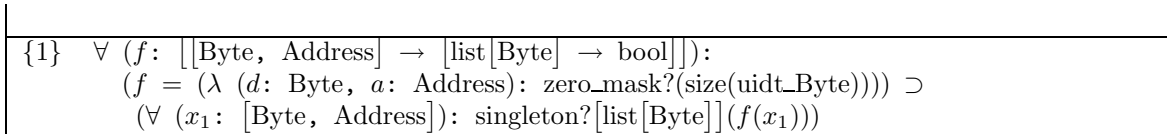
Terse proof for `pod_Byte_TCC3`.
`pod_Byte_TCC3:`



Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of every,
which is trivially true.
This completes the proof of `pod_Byte_TCC3`.
Q.E.D.

C.73.13 Datatype_Model_2.pod_Byte_TCC4

Terse proof for `pod_Byte_TCC4`.
`pod_Byte_TCC4:`



Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pod_Byte_TCC4`.
Q.E.D.

C.73.14 Datatype_Model_2.pod_Byte_TCC5

Terse proof for `pod_Byte_TCC5`.

`pod_Byte_TCC5`:

```
{1}  pod_data_type?[Byte]
       ((#uidt := uidt_Byte,
         to_byte := λ (d: Byte, a: Address): (:1, d:),
         to_mask
           := λ (f: [[Byte, Address] → [list[Byte] → bool]]):
                λ (x1: [Byte, Address]): singleton_elt[list[Byte]](f(x1))
                  (λ (d: Byte, a: Address): zero_mask?(size(uidt_Byte))),
         from_byte
           := λ (bl: list[Byte], a: Address):
                IF length[Byte](bl) = 2 ∧ car[Byte](bl) = 1
                  THEN up[Byte](car[Byte](cdr[Byte](bl)))
                ELSE bottom[Byte]
                ENDIF#))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_Byte_TCC5`.
Q.E.D.

C.74 Proofs for DeclarationStatements1 (statements.pvs)

This theory contains no provable formal statements.

C.75 Proofs for DeclarationStatements2 (statements.pvs)

This theory contains no provable formal statements.

C.76 Proofs for Device_Memory (device_memory.pvs)

This theory contains no provable formal statements.

C.77 Proofs for EqualityExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.78 Proofs for Error_Code_Types (result.pvs)

This theory contains no provable formal statements.

C.79 Proofs for Even_More_List_Props (vfiasco-prelude.pvs)

C.79.1 Even_More_List_Props.every_extend

Terse proof for `every_extend`.

`every_extend:`

$$\{1\}\quad \forall\ (l\colon \text{list}[S],\ P\colon \text{PRED}[T])\colon \text{every}(\lambda\ (t\colon T)\colon P(t))(l) \supset \text{every}(\lambda\ (s\colon S)\colon P(s))(l)$$

Inducting on l on formula 1,

we get 2 subgoals:

`every_extend.1:`

$$\{1\}\quad \forall\ (P\colon \text{PRED}[T])\colon \text{every}(\lambda\ (t\colon T)\colon P(t))(\text{null}) \supset \text{every}(\lambda\ (s\colon S)\colon P(s))(\text{null})$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_extend.1`.

`every_extend.2:`

$$\begin{array}{l}\{1\}\quad \forall\ (\text{cons1\_var}\colon S,\ \text{cons2\_var}\colon \text{list}[S])\colon\\ \quad (\forall\ (P\colon \text{PRED}[T])\colon\\ \quad\quad \text{every}(\lambda\ (t\colon T)\colon P(t))(\text{cons2\_var}) \supset\\ \quad\quad\ \text{every}(\lambda\ (s\colon S)\colon P(s))(\text{cons2\_var}))\\ \quad\ \supset\\ \quad (\forall\ (P\colon \text{PRED}[T])\colon\\ \quad\quad \text{every}(\lambda\ (t\colon T)\colon P(t))(\text{cons}(\text{cons1\_var},\ \text{cons2\_var})) \supset\\ \quad\quad\ \text{every}(\lambda\ (s\colon S)\colon P(s))(\text{cons}(\text{cons1\_var},\ \text{cons2\_var})))\end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_extend.2`.

Q.E.D.

C.80 Proofs for Exception_type (result.pvs)

This theory contains no provable formal statements.

C.81 Proofs for Exception_type_adt (Exception_type_adt.pvs)

This theory contains no provable formal statements.

C.82 Proofs for Exception_type_adt_reduce (Exception_type_adt.pvs)

This theory contains no provable formal statements.

C.83 Proofs for Expand_State (device_memory.pvs)

C.83.1 Expand_State.em_transformers_ok

Terse proof for `em_transformers_ok`.

`em_transformers_ok`:

```
{1}  ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]]):
       transformers_ok?(states, transformers) ⊃
         transformers_ok?(em_lift(states), em_lift(transformers))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_transformers_ok`.
Q.E.D.

C.83.2 Expand_State.em_transformer_invariant

Terse proof for `em_transformer_invariant`.

`em_transformer_invariant`:

```
{1}  ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]]):
       transformer_invariant?(states, transformers) ⊃
         transformer_invariant?(em_lift(states), em_lift(transformers))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_transformer_invariant`.
Q.E.D.

C.83.3 Expand_State.em_lift_singleton

Terse proof for `em_lift_singleton`.

`em_lift_singleton`:

```
{1}  ∀ (q: [State → SuperResult[State]]):
       singleton(em_lift(q)) = em_lift(singleton(q))
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_lift_singleton`.
Q.E.D.

C.84 Proofs for Expand_State2 (device_memory.pvs)

C.84.1 Expand_State2.em_lift_expr_2_super

Terse proof for `em_lift_expr_2_super`.

`em_lift_expr_2_super`:

```
{1}  ∀ (stf: [State → ExprResult[State, Data]]):
       expr_2_super(em_lift(stf)) = em_lift(expr_2_super(stf))
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_lift_expr_2_super`.
Q.E.D.

C.85 Proofs for Expand_State3 (device_memory.pvs)

C.85.1 Expand_State3.em_memory_read_transformers

Terse proof for `em_memory_read_transformers`.

`em_memory_read_transformers`:

```
{1}  ∀ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]],
          pm: Memory_struct[State]):
       (em_pm`memory_read =
          (λ (a: Address): em_lift[State, T, Byte](pm`memory_read(a))))
        ⊃
        memory_read_transformers(em_pm, addresses) =
         em_lift(memory_read_transformers(pm, addresses))
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Expanding the definition of memory_read_transformers,
Replacing using formula -1,
Hiding formulas: -1,
Expanding the definition of em_lift,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`em_memory_read_transformers.1`:

```
{-1}  ∃ (a: Address):
        addresses′(a) ∧ x′ = expr_2_super(em_lift[State, T, Byte](pm′`memory_read(a)))
{1}  ∃ (p:
          (λ (q: [State → SuperResult[State]]):
             ∃ (a: Address):
               addresses′(a) ∧ q = expr_2_super(memory_read(pm′)(a)))):
       x′ = em_lift(p)
```

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (expr_2_super(pm!1`memory_read(a!1))),
we get 2 subgoals:

em_memory_read_transformers.1.1:

```
{-1}  addresses'(a')
{-2}  x' = expr_2_super(em_lift[State, T, Byte](pm'`memory_read(a')))
------------------------------------------------------------------------
{1}   x' = em_lift(expr_2_super(pm'`memory_read(a')))
```

Rewriting using em_lift_expr_2_super[State, T, Byte], matching in *,
This completes the proof of em_memory_read_transformers.1.1.
em_memory_read_transformers.1.2:

```
{-1}  addresses'(a')
{-2}  x' = expr_2_super(em_lift[State, T, Byte](pm'`memory_read(a')))
------------------------------------------------------------------------
{1}   ∃ (a: Address):
        addresses'(a) ∧
         expr_2_super[State, Byte](pm'`memory_read(a')) =
           expr_2_super[State, Byte](memory_read(pm')(a))
```

Instantiating quantified variables,
This completes the proof of em_memory_read_transformers.1.2.
em_memory_read_transformers.2:

```
{-1}  ∃ (p:
            (λ (q: [State → SuperResult[State]]):
                 ∃ (a: Address):
                   addresses'(a) ∧ q = expr_2_super(memory_read(pm')(a)))):
        x' = em_lift(p)
------------------------------------------------------------------------
{1}   ∃ (a: Address):
        addresses'(a) ∧ x' = expr_2_super(em_lift[State, T, Byte](pm'`memory_read(a)))
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Rewriting using em_lift_expr_2_super[State, T, Byte], matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of em_memory_read_transformers.2.
Q.E.D.

C.85.2 Expand_State3.em_memory_write_transformers

Terse proof for em_memory_write_transformers.
em_memory_write_transformers:

```
------------------------------------------------------------------------
{1}   ∀ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]],
           pm: Memory_struct[State]):
        (em_pm`memory_write =
           (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm`memory_write(a, b))))
         ⊃
         memory_write_transformers(em_pm, addresses) =
          em_lift(memory_write_transformers(pm, addresses))
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Expanding the definition of memory_write_transformers,
Replacing using formula -1,
Hiding formulas: -1,
Expanding the definition of em_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`em_memory_write_transformers.1:`

```
{-1}  ∃ (a: Address, b: Byte):
        addresses′(a) ∧
          x′ = expr_2_super(em_lift[State, T, Unit](pm′‘memory_write(a, b)))
{1}   ∃ (p:
           (λ (q: [State → SuperResult[State]]):
                ∃ (a: Address, b: Byte):
                  addresses′(a) ∧ q = expr_2_super(memory_write(pm′)(a, b)))):
        x′ = em_lift(p)
```

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (expr_2_super(pm!1‘memory_write(a!1, b!1))),
we get 2 subgoals:
`em_memory_write_transformers.1.1:`

```
{-1}  b′ < max_byte
{-2}  addresses′(a′)
{-3}  x′ = expr_2_super(em_lift[State, T, Unit](pm′‘memory_write(a′, b′)))
{1}   x′ = em_lift(expr_2_super(pm′‘memory_write(a′, b′)))
```

Rewriting using em_lift_expr_2_super, matching in *,
This completes the proof of `em_memory_write_transformers.1.1`.
`em_memory_write_transformers.1.2:`

```
{-1}  b′ < max_byte
{-2}  addresses′(a′)
{-3}  x′ = expr_2_super(em_lift[State, T, Unit](pm′‘memory_write(a′, b′)))
{1}   ∃ (a: Address, b: Byte):
        addresses′(a) ∧
          expr_2_super[State, Unit](pm′‘memory_write(a′, b′)) =
            expr_2_super[State, Unit](memory_write(pm′)(a, b))
```

Instantiating quantified variables,
This completes the proof of `em_memory_write_transformers.1.2`.
`em_memory_write_transformers.2:`

```
{-1}  ∃ (p:
           (λ (q: [State → SuperResult[State]]):
                ∃ (a: Address, b: Byte):
                  addresses′(a) ∧ q = expr_2_super(memory_write(pm′)(a, b)))):
        x′ = em_lift(p)
{1}   ∃ (a: Address, b: Byte):
        addresses′(a) ∧
          x′ = expr_2_super(em_lift[State, T, Unit](pm′‘memory_write(a, b)))
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Rewriting using em_lift_expr_2_super, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `em_memory_write_transformers.2`.
Q.E.D.

C.85.3 Expand_State3.em_unchanged_memory_invariant

Terse proof for `em_unchanged_memory_invariant`.

`em_unchanged_memory_invariant`:

{1} ∀ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]],
pm: Memory_struct[State], states: PRED[State],
transformers: PRED[[State → SuperResult[State]]]):
(em_pm‘memory_read =
(λ (a: Address): em_lift[State, T, Byte](pm‘memory_read(a))))
∧ unchanged_memory_invariant?(pm, states, transformers, addresses)
⊃ unchanged_memory_invariant?(em_pm, em_lift(states), em_lift(transformers), addresses)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `em_unchanged_memory_invariant`.

Q.E.D.

C.85.4 Expand_State3.em_changed_memory_invariant

Terse proof for `em_changed_memory_invariant`.

`em_changed_memory_invariant`:

{1} ∀ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]],
pm: Memory_struct[State], states: PRED[State]):
(em_pm‘memory_read =
(λ (a: Address): em_lift[State, T, Byte](pm‘memory_read(a))))
∧
(em_pm‘memory_write =
(λ (a: Address, b: Byte): em_lift[State, T, Unit](pm‘memory_write(a, b))))
∧ changed_memory_invariant?(pm, states, addresses)
⊃ changed_memory_invariant?(em_pm, em_lift(states), addresses)

Expanding the definition of changed_memory_invariant?,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`em_changed_memory_invariant.1:`

```
{-1}  (em_pm'`memory_read =
         (λ (a: Address): em_lift[State, T, Byte](pm'`memory_read(a))))
{-2}  (em_pm'`memory_write =
         (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm'`memory_write(a, b))))
{-3}  transformer_invariant?(states', memory_read_transformers(pm', addresses'))
{-4}  transformer_invariant?(states', memory_write_transformers(pm', addresses'))
{-5}  ∀ (s: State, a: Address, b: Byte):
        states'(s) ∧
         addresses'(a) ∧
          OK?(memory_write(pm')(a, b)(s)) ∧
           OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s))))
        ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b
-------------------------------------------------------------------------
{1}   ∀ (s: Expand_state[State, T], a: Address, b: Byte):
        em_lift(states')(s) ∧
         addresses'(a) ∧
          OK?(memory_write(em_pm')(a, b)(s)) ∧
           OK?(memory_read(em_pm')(a)(state(memory_write(em_pm')(a, b)(s))))
         ⊃ data(memory_read(em_pm')(a)(state(memory_write(em_pm')(a, b)(s)))) = b
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `em_changed_memory_invariant.1`.

`em_changed_memory_invariant.2:`

```
{-1}  (em_pm'`memory_read =
         (λ (a: Address): em_lift[State, T, Byte](pm'`memory_read(a))))
{-2}  (em_pm'`memory_write =
         (λ (a: Address, b: Byte): em_lift[State, T, Unit](pm'`memory_write(a, b))))
{-3}  transformer_invariant?(states', memory_read_transformers(pm', addresses'))
{-4}  transformer_invariant?(states', memory_write_transformers(pm', addresses'))
{-5}  ∀ (s: State, a: Address, b: Byte):
        states'(s) ∧
         addresses'(a) ∧
          OK?(memory_write(pm')(a, b)(s)) ∧
           OK?(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s))))
        ⊃ data(memory_read(pm')(a)(state(memory_write(pm')(a, b)(s)))) = b
-------------------------------------------------------------------------
{1}   transformer_invariant?(em_lift(states'), memory_write_transformers(em_pm', addresses'))
```

Keeping (-2 -4 1) and hiding *,

Using lemma em_memory_write_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -2,

Rewriting using em_transformer_invariant, matching in *,

This completes the proof of `em_changed_memory_invariant.2`.

`em_changed_memory_invariant.3`:

{-1} (em_pm′‘memory_read =
(λ (a: Address): em_lift[State, T, Byte](pm′‘memory_read(a))))
{-2} (em_pm′‘memory_write =
(λ (a: Address, b: Byte): em_lift[State, T, Unit](pm′‘memory_write(a, b))))
{-3} transformer_invariant?(states′, memory_read_transformers(pm′, addresses′))
{-4} transformer_invariant?(states′, memory_write_transformers(pm′, addresses′))
{-5} ∀ (s: State, a: Address, b: Byte):
states′(s) ∧
addresses′(a) ∧
OK?(memory_write(pm′)(a, b)(s)) ∧
OK?(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s))))
⊃ data(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s)))) = b
---
{1} transformer_invariant?(em_lift(states′), memory_read_transformers(em_pm′, addresses′))

Keeping (-1 -3 1) and hiding *,
Using lemma em_memory_read_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Rewriting using em_transformer_invariant, matching in *,
This completes the proof of `em_changed_memory_invariant.3`.
Q.E.D.

C.85.5 Expand_State3.em_const_unchanged_memory_invariant_TCC1

Terse proof for `em_const_unchanged_memory_invariant_TCC1`.

`em_const_unchanged_memory_invariant_TCC1`:

{1} ∀ (em_pm: Memory_struct[Expand_state[State, T]], pm: Memory_struct[State],
q: [Expand_state[State, T] → SuperResult[Expand_state[State, T]]]):
(em_pm‘memory_read =
(λ (a: Address): em_lift[State, T, Byte](pm‘memory_read(a))))
⊃
(∀ (s: Expand_state[State, T]):
has_next_state(q(s)) ⊃
OK?[Expand_state[State, T]](q(s)) ∨
abnormal?[Expand_state[State, T]](q(s)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_const_unchanged_memory_invariant_TCC1`.
Q.E.D.

C.85.6 Expand_State3.em_const_unchanged_memory_invariant

Terse proof for `em_const_unchanged_memory_invariant`.

`em_const_unchanged_memory_invariant:`

```
{1}  ∀ (addresses: PRED[Address], em_pm: Memory_struct[Expand_state[State, T]],
          pm: Memory_struct[State], states: PRED[State],
          q: [Expand_state[State, T] → SuperResult[Expand_state[State, T]]]):
       (em_pm`memory_read =
          (λ (a: Address): em_lift[State, T, Byte](pm`memory_read(a))))
        ∧
        (∀ (s: Expand_state[State, T]):
            has_next_state(q(s)) ⊃ state(q(s))`state = s`state)
        ⊃ unchanged_memory_invariant?(em_pm, em_lift(states), singleton(q), addresses)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_const_unchanged_memory_invariant`.
Q.E.D.

C.85.7 Expand_State3.em_side_effect_content_unchanged

Terse proof for `em_side_effect_content_unchanged`.

`em_side_effect_content_unchanged:`

```
{1}  ∀ (addresses: PRED[Address],
          se_transformer:
            [[Address, list[Byte], bool] → [State → ExprResult[State, list[Byte]]]],
          states: PRED[State]):
       side_effect_content_unchanged(addresses, states, se_transformer) ⊃
        side_effect_content_unchanged(addresses, em_lift(states),
                                      λ (a: Address, bl: list[Byte], cp: bool):
                                        em_lift[State, T, list[Byte]]
                                          (se_transformer(a, bl, cp)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `em_side_effect_content_unchanged`.
Q.E.D.

C.86 Proofs for ExprResult (result.pvs)

This theory contains no provable formal statements.

C.87 Proofs for ExprResult_adt (ExprResult_adt.pvs)

This theory contains no provable formal statements.

C.88 Proofs for ExprResult_adt_map (ExprResult_adt.pvs)

This theory contains no provable formal statements.

C.89 Proofs for ExprResult_adt_reduce (ExprResult_adt.pvs)

This theory contains no provable formal statements.

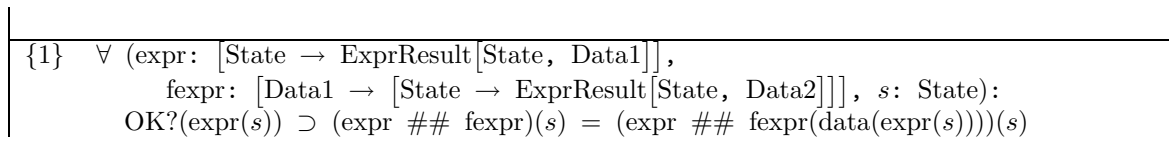
C.90 Proofs for ExpressionStatements (statements.pvs)

This theory contains no provable formal statements.

C.91 Proofs for Expression_Composition_Rewrites (state-transformer.pvs)

C.91.1 Expression_Composition_Rewrites.comp_eval_if_ok_fexpr

Terse proof for `comp_eval_if_ok_fexpr`.
`comp_eval_if_ok_fexpr:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_eval_if_ok_fexpr`.
Q.E.D.

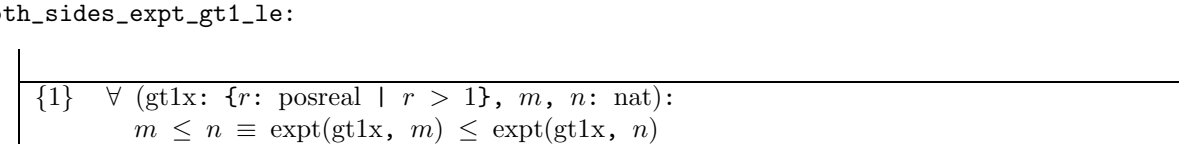
C.92 Proofs for Expressions (expressions.pvs)

This theory contains no provable formal statements.

C.93 Proofs for Expt_Lemmas (vfiasco-prelude.pvs)

C.93.1 Expt_Lemmas.both_sides_expt_gt1_le

Terse proof for `both_sides_expt_gt1_le`.
`both_sides_expt_gt1_le:`



Repeatedly Skolemizing and flattening,
Using lemma both_sides_expt_gt1_le_aux,
Expanding the definition of expt,
Using lemma both_sides_times_pos_le2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `both_sides_expt_gt1_le`.
Q.E.D.

C.93.2 Expt_Lemmas.expt_plus

Terse proof for `expt_plus`.
`expt_plus`:

{1} ∀ (n0x: nzreal, m, n: nat):
expt(n0x, m) × expt(n0x, n) = expt(n0x, $m+n$)

Repeatedly Skolemizing and flattening,
Using lemma expt_plus_aux,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `expt_plus`.
Q.E.D.

C.94 Proofs for Extended_Real (vfiasco-prelude.pvs)

C.94.1 Extended_Real.number_field_extended

Terse proof for `number_field_extended`.
`number_field_extended`:

{1} ∀ (x: number_field): extended_real?(x)

Repeatedly Skolemizing and flattening,
Rewriting using extended_real_superset_of_number_fields, matching in *,
This completes the proof of `number_field_extended`.
Q.E.D.

C.94.2 Extended_Real.er_plus_TCC1

Terse proof for `er_plus_TCC1`.
`er_plus_TCC1`:

{1} ∃ (x_1: ⌊⌊extended_real, extended_real⌋ → extended_real⌋): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (x, y: extended_real): x),
which is trivially true.
This completes the proof of `er_plus_TCC1`.
Q.E.D.

C.94.3 Extended_Real.er_neg_TCC1

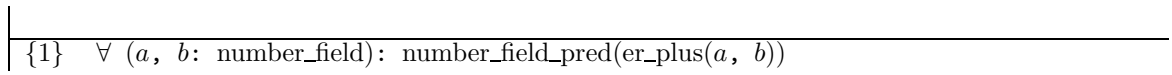
Terse proof for `er_neg_TCC1`.
`er_neg_TCC1`:

{1} ∃ (x_1: ⌊extended_real → extended_real⌋): TRUE

Instantiating the top quantifier in 1 with the terms: (λ (x: extended_real): x),
which is trivially true.
This completes the proof of `er_neg_TCC1`.
Q.E.D.

C.94.4 Extended_Real.er_plus_number_field

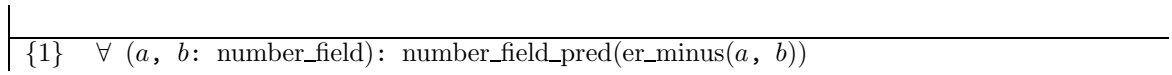
Terse proof for `er_plus_number_field`.
`er_plus_number_field`:



Repeatedly Skolemizing and flattening,
Rewriting using er_plus_ax, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `er_plus_number_field`.
Q.E.D.

C.94.5 Extended_Real.er_minus_number_field

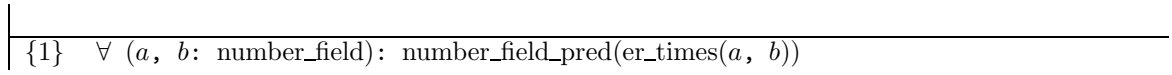
Terse proof for `er_minus_number_field`.
`er_minus_number_field`:



Repeatedly Skolemizing and flattening,
Rewriting using er_minus_ax, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `er_minus_number_field`.
Q.E.D.

C.94.6 Extended_Real.er_times_number_field

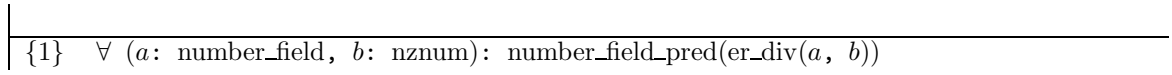
Terse proof for `er_times_number_field`.
`er_times_number_field`:



Repeatedly Skolemizing and flattening,
Rewriting using er_times_ax, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `er_times_number_field`.
Q.E.D.

C.94.7 Extended_Real.er_div_number_field

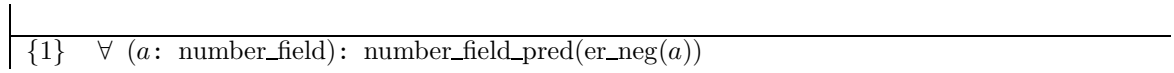
Terse proof for `er_div_number_field`.
`er_div_number_field`:



Repeatedly Skolemizing and flattening,
Rewriting using er_div_ax, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `er_div_number_field`.
Q.E.D.

C.94.8 Extended_Real.er_neg_number_field

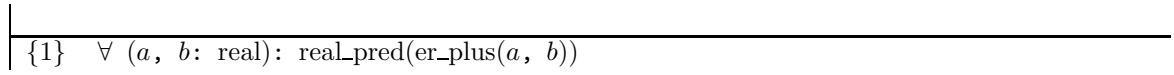
Terse proof for `er_neg_number_field`.
`er_neg_number_field:`



Repeatedly Skolemizing and flattening,
Rewriting using er_neg_ax, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `er_neg_number_field`.
Q.E.D.

C.94.9 Extended_Real.er_plus_real

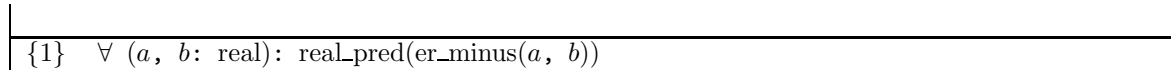
Terse proof for `er_plus_real`.
`er_plus_real:`



Repeatedly Skolemizing and flattening,
Rewriting using er_plus_ax, matching in *,
Rewriting using closed_plus, matching in *,
This completes the proof of `er_plus_real`.
Q.E.D.

C.94.10 Extended_Real.er_minus_real

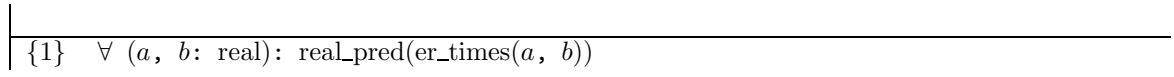
Terse proof for `er_minus_real`.
`er_minus_real:`



Repeatedly Skolemizing and flattening,
Rewriting using er_minus_ax, matching in *,
Rewriting using closed_minus, matching in *,
This completes the proof of `er_minus_real`.
Q.E.D.

C.94.11 Extended_Real.er_times_real

Terse proof for `er_times_real`.
`er_times_real:`



Repeatedly Skolemizing and flattening,
Rewriting using er_times_ax, matching in *,
Rewriting using closed_times, matching in *,
This completes the proof of `er_times_real`.
Q.E.D.

C.94.12 Extended_Real.er_div_real

Terse proof for `er_div_real`.
`er_div_real`:

|
|---
| {1} ∀ (a: real, b: nzreal): real_pred(er_div(a, b))

Repeatedly Skolemizing and flattening,
Rewriting using er_div_ax, matching in *,
Rewriting using closed_divides, matching in *,
This completes the proof of `er_div_real`.
Q.E.D.

C.94.13 Extended_Real.er_neg_real

Terse proof for `er_neg_real`.
`er_neg_real`:

|
|---
| {1} ∀ (a: real): real_pred(er_neg(a))

Repeatedly Skolemizing and flattening,
Rewriting using er_neg_ax, matching in *,
Rewriting using closed_neg, matching in *,
This completes the proof of `er_neg_real`.
Q.E.D.

C.94.14 Extended_Real.er_plus_rational

Terse proof for `er_plus_rational`.
`er_plus_rational`:

|
|---
| {1} ∀ (a, b: rational): rational_pred(er_plus(a, b))

Repeatedly Skolemizing and flattening,
Rewriting using er_plus_ax, matching in *,
Rewriting using closed_plus, matching in *,
This completes the proof of `er_plus_rational`.
Q.E.D.

C.94.15 Extended_Real.er_minus_rational

Terse proof for `er_minus_rational`.
`er_minus_rational`:

|
|---
| {1} ∀ (a, b: rational): rational_pred(er_minus(a, b))

Repeatedly Skolemizing and flattening,
Rewriting using er_minus_ax, matching in *,
Rewriting using closed_minus, matching in *,
This completes the proof of `er_minus_rational`.
Q.E.D.

C.94.16 Extended_Real.er_times_rational

Terse proof for `er_times_rational`.
`er_times_rational:`

```
|
|-------------------------------------------------------------------------------
| {1}  ∀ (a, b: rational): rational_pred(er_times(a, b))
```

Repeatedly Skolemizing and flattening,
Rewriting using er_times_ax, matching in *,
Rewriting using closed_times, matching in *,
This completes the proof of `er_times_rational`.
Q.E.D.

C.94.17 Extended_Real.er_div_rational

Terse proof for `er_div_rational`.
`er_div_rational:`

```
|
|-------------------------------------------------------------------------------
| {1}  ∀ (a: rational, b: nzrat): rational_pred(er_div(a, b))
```

Repeatedly Skolemizing and flattening,
Rewriting using er_div_ax, matching in *,
Rewriting using closed_divides, matching in *,
This completes the proof of `er_div_rational`.
Q.E.D.

C.94.18 Extended_Real.er_neg_rational

Terse proof for `er_neg_rational`.
`er_neg_rational:`

```
|
|-------------------------------------------------------------------------------
| {1}  ∀ (a: rational): rational_pred(er_neg(a))
```

Repeatedly Skolemizing and flattening,
Rewriting using er_neg_ax, matching in *,
Rewriting using closed_neg, matching in *,
This completes the proof of `er_neg_rational`.
Q.E.D.

C.94.19 Extended_Real.er_plus_integer

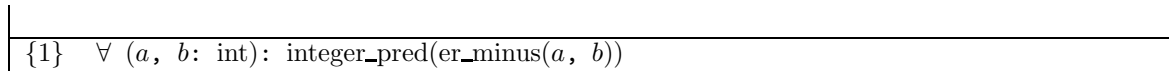
Terse proof for `er_plus_integer`.
`er_plus_integer:`

```
|
|-------------------------------------------------------------------------------
| {1}  ∀ (a, b: int): integer_pred(er_plus(a, b))
```

Repeatedly Skolemizing and flattening,
Rewriting using er_plus_ax, matching in *,
Rewriting using closed_plus, matching in *,
This completes the proof of `er_plus_integer`.
Q.E.D.

C.94.20 Extended_Real.er_minus_integer

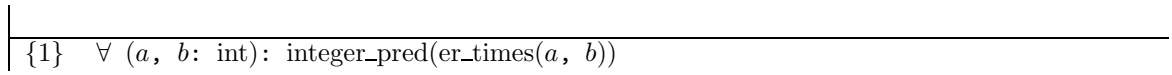
Terse proof for `er_minus_integer`.
`er_minus_integer`:



Repeatedly Skolemizing and flattening,
Rewriting using er_minus_ax, matching in *,
Rewriting using closed_minus, matching in *,
This completes the proof of `er_minus_integer`.
Q.E.D.

C.94.21 Extended_Real.er_times_integer

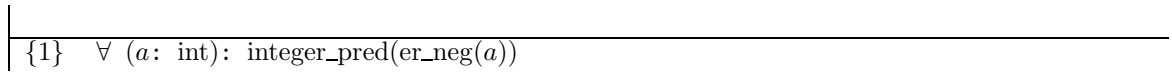
Terse proof for `er_times_integer`.
`er_times_integer`:



Repeatedly Skolemizing and flattening,
Rewriting using er_times_ax, matching in *,
Rewriting using closed_times, matching in *,
This completes the proof of `er_times_integer`.
Q.E.D.

C.94.22 Extended_Real.er_neg_integer

Terse proof for `er_neg_integer`.
`er_neg_integer`:

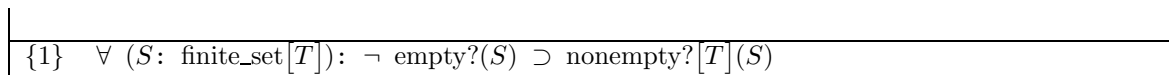


Repeatedly Skolemizing and flattening,
Rewriting using er_neg_ax, matching in *,
Rewriting using closed_neg, matching in *,
This completes the proof of `er_neg_integer`.
Q.E.D.

C.95 Proofs for Finite_Set_Reduce (vfiasco-prelude.pvs)

C.95.1 Finite_Set_Reduce.list_of_finite_set_TCC1

Terse proof for `list_of_finite_set_TCC1`.
`list_of_finite_set_TCC1`:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `list_of_finite_set_TCC1`.
Q.E.D.

C.95.2 Finite_Set_Reduce.list_of_finite_set_TCC2

Terse proof for `list_of_finite_set_TCC2`.

`list_of_finite_set_TCC2:`

$$\{1\}\quad \forall\ (S\colon \text{finite\_set}[T])\colon \neg\ \text{empty?}(S)\ \supset\ \text{card}[T](\text{rest}[T](S))\ <\ \text{card}[T](S)$$

Repeatedly Skolemizing and flattening,
Rewriting using card_rest, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `list_of_finite_set_TCC2`.
Q.E.D.

C.95.3 Finite_Set_Reduce.length_list_of_finite_set

Terse proof for `length_list_of_finite_set`.

`length_list_of_finite_set:`

$$\{1\}\quad \forall\ (S\colon \text{finite\_set}[T])\colon \text{length}(\text{list\_of\_finite\_set}(S))\ =\ \text{card}(S)$$

Case splitting on $\forall\ (i\colon \text{nat},\ S\colon \text{finite\_set}[T])\colon\ i = \text{card}(S) \supset \text{length}(\text{list\_of\_finite\_set}(S)) = \text{card}(S)$,
we get 2 subgoals:

`length_list_of_finite_set.1:`

$$\begin{array}{l}\{\text{-}1\}\quad \forall\ (i\colon \text{nat},\ S\colon \text{finite\_set}[T])\colon \\ \qquad i\ =\ \text{card}(S)\ \supset\ \text{length}(\text{list\_of\_finite\_set}(S))\ =\ \text{card}(S) \\ \hline \{1\}\quad \forall\ (S\colon \text{finite\_set}[T])\colon\ \text{length}(\text{list\_of\_finite\_set}(S))\ =\ \text{card}(S)\end{array}$$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: $\text{card}(S')$, S',
This completes the proof of `length_list_of_finite_set.1`.

`length_list_of_finite_set.2:`

$$\begin{array}{l}\{1\}\quad \forall\ (i\colon \text{nat},\ S\colon \text{finite\_set}[T])\colon \\ \qquad i\ =\ \text{card}(S)\ \supset\ \text{length}(\text{list\_of\_finite\_set}(S))\ =\ \text{card}(S) \\ \{2\}\quad \forall\ (S\colon \text{finite\_set}[T])\colon\ \text{length}(\text{list\_of\_finite\_set}(S))\ =\ \text{card}(S)\end{array}$$

Hiding formulas: 2,
Inducting on i on formula 1,
we get 2 subgoals:

`length_list_of_finite_set.2.1:`

$$\{1\}\quad \forall\ (S\colon \text{finite\_set}[T])\colon\ 0\ =\ \text{card}(S)\ \supset\ \text{length}(\text{list\_of\_finite\_set}(S))\ =\ \text{card}(S)$$

Repeatedly Skolemizing and flattening,
Using lemma $\text{card\_empty?}[T]$,
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_list_of_finite_set.2.1`.

`length_list_of_finite_set.2.2:`

{1} ∀ j:
(∀ (S: finite_set[T]):
j = card(S) ⊃ length(list_of_finite_set(S)) = card(S))
⊃
(∀ (S: finite_set[T]):
$j+1$ = card(S) ⊃ length(list_of_finite_set(S)) = card(S))

Repeatedly Skolemizing and flattening,
Using lemma card_empty?[T],
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of list_of_finite_set,
Expanding the definition of length,
Instantiating quantified variables,
Rewriting using card_rest, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `length_list_of_finite_set.2.2`.
Q.E.D.

C.95.4 Finite_Set_Reduce.member_list_of_finite_set

Terse proof for `member_list_of_finite_set`.

`member_list_of_finite_set:`

{1} ∀ (S: finite_set[T], x: T): ($x \in S$) = member(x, list_of_finite_set(S))

Case splitting on ∀ (S: finite_set[T], k: nat, x: T): k = card(S) ⊃ (member(x, S) = member(x, list_of_finite_set(S))),
we get 2 subgoals:

`member_list_of_finite_set.1:`

{-1} ∀ (S: finite_set[T], k: nat, x: T):
k = card(S) ⊃ (($x \in S$) = member(x, list_of_finite_set(S)))

{1} ∀ (S: finite_set[T], x: T): ($x \in S$) = member(x, list_of_finite_set(S))

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating quantified variables,
This completes the proof of `member_list_of_finite_set.1`.

`member_list_of_finite_set.2:`

{1} ∀ (S: finite_set[T], k: nat, x: T):
k = card(S) ⊃ (($x \in S$) = member(x, list_of_finite_set(S)))
{2} ∀ (S: finite_set[T], x: T): ($x \in S$) = member(x, list_of_finite_set(S))

Hiding formulas: 2,
Inducting on k on formula 1,
we get 2 subgoals:

`member_list_of_finite_set.2.1:`

{1} ∀ (S: finite_set[T], x: T):
0 = card(S) ⊃ ((x ∈ S) = member(x, list_of_finite_set(S)))

Repeatedly Skolemizing and flattening,
Using lemma card_empty?[T],
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `member_list_of_finite_set.2.1`.
`member_list_of_finite_set.2.2:`

{1} ∀ j:
(∀ (S: finite_set[T], x: T):
j = card(S) ⊃ ((x ∈ S) = member(x, list_of_finite_set(S))))
⊃
(∀ (S: finite_set[T], x: T):
j + 1 = card(S) ⊃ ((x ∈ S) = member(x, list_of_finite_set(S))))

Repeatedly Skolemizing and flattening,
Using lemma card_empty?[T],
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of list_of_finite_set,
Expanding the definition of member,
Case splitting on x!1 = choose(S!1),
we get 2 subgoals:
`member_list_of_finite_set.2.2.1:`

{-1} x′ = choose(S′)
{-2} is_finite[T](S′)
{-3} j′ ≥ 0
{-4} ∀ (S: finite_set[T], x: T):
j′ = card(S) ⊃ ((x ∈ S) = member(x, list_of_finite_set(S)))
{-5} 1 + j′ = card(S′)

{1} empty?(S′)
{2} ((x′ ∈ S′) = (x′ = choose(S′) ∨ member(x′, list_of_finite_set(rest(S′)))))

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of member,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `member_list_of_finite_set.2.2.1`.
`member_list_of_finite_set.2.2.2:`

{-1} is_finite[T](S′)
{-2} j′ ≥ 0
{-3} ∀ (S: finite_set[T], x: T):
j′ = card(S) ⊃ ((x ∈ S) = member(x, list_of_finite_set(S)))
{-4} 1 + j′ = card(S′)

{1} x′ = choose(S′)
{2} empty?(S′)
{3} ((x′ ∈ S′) = (x′ = choose(S′) ∨ member(x′, list_of_finite_set(rest(S′)))))

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -3 with the terms: rest(S'), x',
Installing automatic rewrites from: card_rest
Simplifying, rewriting, and recording with decision procedures,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`member_list_of_finite_set.2.2.2.1:`

{-1}	is_finite$[T]$(S')
{-2}	$j' \geq 0$
{-3}	(x' $\in$ rest(S'))
{-4}	member(x', list_of_finite_set(rest(S')))
{-5}	$1 + j' = \text{card}(S')$
{1}	$x' = \text{choose}(S')$
{2}	empty?(S')
{3}	(x' $\in$ S')

Using lemma rest_member,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `member_list_of_finite_set.2.2.2.1`.
`member_list_of_finite_set.2.2.2.2:`

{-1}	is_finite$[T]$(S')
{-2}	$j' \geq 0$
{-3}	$1 + j' = \text{card}(S')$
{-4}	(x' $\in$ S')
{1}	$x' = \text{choose}(S')$
{2}	empty?(S')
{3}	(x' $\in$ rest(S'))
{4}	member(x', list_of_finite_set(rest(S')))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `member_list_of_finite_set.2.2.2.2`.
Q.E.D.

C.95.5 Finite_Set_Reduce.unique_member_list_of_finite_set

Terse proof for `unique_member_list_of_finite_set`.
`unique_member_list_of_finite_set:`

{1}	$\forall$ (S: finite_set$[T]$, i, j: below(length(list_of_finite_set(S)))): nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) $\supset$ $i = j$

Case splitting on $\forall$ (S: finite_set$[T]$, k: nat, i, j: below(length(list_of_finite_set(S)))): k = card(S) $\wedge$ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) $\supset$ $i = j$,
we get 2 subgoals:
`unique_member_list_of_finite_set.1:`

{-1}	$\forall$ (S: finite_set$[T]$, k: nat, i, j: below(length(list_of_finite_set(S)))): k = card(S) $\wedge$ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) $\supset$ $i = j$
{1}	$\forall$ (S: finite_set$[T]$, i, j: below(length(list_of_finite_set(S)))): nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) $\supset$ $i = j$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -4 with the terms: S', card(S'), i', j',

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unique_member_list_of_finite_set.1`.

`unique_member_list_of_finite_set.2:`

```
{1}  ∀ (S: finite_set[T], k: nat, i, j: below(length(list_of_finite_set(S)))):
       k = card(S) ∧ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) ⊃
        i = j
{2}  ∀ (S: finite_set[T], i, j: below(length(list_of_finite_set(S)))):
       nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) ⊃ i = j
```

Hiding formulas: 2,

Inducting on k on formula 1,

we get 2 subgoals:

`unique_member_list_of_finite_set.2.1:`

```
{1}  ∀ (S: finite_set[T], i, j: below(length(list_of_finite_set(S)))):
       0 = card(S) ∧ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) ⊃
        i = j
```

Repeatedly Skolemizing and flattening,

Using lemma card_empty?[T],

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unique_member_list_of_finite_set.2.1`.

`unique_member_list_of_finite_set.2.2:`

```
{1}  ∀ (j_1: nat):
       (∀ (S: finite_set[T], i, j: below(length(list_of_finite_set(S)))):
            j_1 = card(S) ∧
             nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j)
             ⊃ i = j)
        ⊃
       (∀ (S: finite_set[T], i, j: below(length(list_of_finite_set(S)))):
            j_1 + 1 = card(S) ∧
             nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j)
             ⊃ i = j)
```

Repeatedly Skolemizing and flattening,

Using lemma card_empty?[T],

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of list_of_finite_set,

Case splitting on i!1 = 0,

we get 2 subgoals:

`unique_member_list_of_finite_set.2.2.1:`

{-1} $i' = 0$
{-2} is_finite$[T](S')$
{-3} $i' <$ length(list_of_finite_set(S'))
{-4} $j'' <$ length(list_of_finite_set(S'))
{-5} $j' \geq 0$
{-6} $\forall$ (S: finite_set$[T]$, i, j: below(length(list_of_finite_set(S)))):
$j' =$ card(S) $\wedge$ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j)
$\supset i = j$
{-7} $1 + j' =$ card(S')
{-8} nth(cons(choose(S'), list_of_finite_set(rest(S'))), i') =
nth(cons(choose(S'), list_of_finite_set(rest(S'))), j'')

{1} empty?(S')
{2} $i' = j''$

Expanding the definition of nth,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -6,
Using lemma member_list_of_finite_set,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`unique_member_list_of_finite_set.2.2.1.1:`

{-1} (choose(S') $\in$ rest(S'))
{-2} member(choose(S'), list_of_finite_set(rest(S')))
{-3} $i' = 0$
{-4} is_finite$[T](S')$
{-5} $i' <$ length(list_of_finite_set(S'))
{-6} $j'' <$ length(list_of_finite_set(S'))
{-7} $j' \geq 0$
{-8} $1 + j' =$ card(S')
{-9} choose(S') = nth(list_of_finite_set(rest(S')), $j'' - 1$)

{1} empty?(S')
{2} $i' = j''$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unique_member_list_of_finite_set.2.2.1.1`.
`unique_member_list_of_finite_set.2.2.1.2:`

{-1} $i' = 0$
{-2} is_finite$[T](S')$
{-3} $i' <$ length(list_of_finite_set(S'))
{-4} $j'' <$ length(list_of_finite_set(S'))
{-5} $j' \geq 0$
{-6} $1 + j' =$ card(S')
{-7} choose(S') = nth(list_of_finite_set(rest(S')), $j'' - 1$)

{1} (choose(S') $\in$ rest(S'))
{2} member(choose(S'), list_of_finite_set(rest(S')))
{3} empty?(S')
{4} $i' = j''$

Hiding formulas: 1,
Using lemma nth_member$[T]$,
Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unique_member_list_of_finite_set.2.2.1.2`.

`unique_member_list_of_finite_set.2.2.2:`

{-1}	is_finite$[T](S')$
{-2}	$i' <$ length(list_of_finite_set(S'))
{-3}	$j'' <$ length(list_of_finite_set(S'))
{-4}	$j' \geq 0$
{-5}	$\forall$ (S: finite_set$[T]$, i, j: below(length(list_of_finite_set(S)))): $j' =$ card(S) $\wedge$ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) $\supset$ $i = j$
{-6}	$1 + j' =$ card(S')
{-7}	nth(cons(choose(S'), list_of_finite_set(rest(S'))), i') = nth(cons(choose(S'), list_of_finite_set(rest(S'))), j'')
{1}	$i' = 0$
{2}	empty?(S')
{3}	$i' = j''$

Case splitting on j!2 = 0,

we get 2 subgoals:

`unique_member_list_of_finite_set.2.2.2.1:`

{-1}	$j'' = 0$
{-2}	is_finite$[T](S')$
{-3}	$i' <$ length(list_of_finite_set(S'))
{-4}	$j'' <$ length(list_of_finite_set(S'))
{-5}	$j' \geq 0$
{-6}	$\forall$ (S: finite_set$[T]$, i, j: below(length(list_of_finite_set(S)))): $j' =$ card(S) $\wedge$ nth(list_of_finite_set(S), i) = nth(list_of_finite_set(S), j) $\supset$ $i = j$
{-7}	$1 + j' =$ card(S')
{-8}	nth(cons(choose(S'), list_of_finite_set(rest(S'))), i') = nth(cons(choose(S'), list_of_finite_set(rest(S'))), j'')
{1}	$i' = 0$
{2}	empty?(S')
{3}	$i' = j''$

Hiding formulas: -6,

Expanding the definition of nth,

Simplifying, rewriting, and recording with decision procedures,

Using lemma member_list_of_finite_set,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unique_member_list_of_finite_set.2.2.2.1.1:`

{-1} (choose(S') ∈ rest(S'))
{-2} member(choose(S'), list_of_finite_set(rest(S')))
{-3} $j'' = 0$
{-4} is_finite[T](S')
{-5} $i' < \text{length}(\text{list\_of\_finite\_set}(S'))$
{-6} $j'' < \text{length}(\text{list\_of\_finite\_set}(S'))$
{-7} $j' \geq 0$
{-8} $1 + j' = \text{card}(S')$
{-9} nth(list_of_finite_set(rest(S')), $i' - 1$) = choose(S')

{1} $i' = 0$
{2} empty?(S')
{3} $i' = j''$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unique_member_list_of_finite_set.2.2.2.1.1`.
`unique_member_list_of_finite_set.2.2.2.1.2:`

{-1} $j'' = 0$
{-2} is_finite[T](S')
{-3} $i' < \text{length}(\text{list\_of\_finite\_set}(S'))$
{-4} $j'' < \text{length}(\text{list\_of\_finite\_set}(S'))$
{-5} $j' \geq 0$
{-6} $1 + j' = \text{card}(S')$
{-7} nth(list_of_finite_set(rest(S')), $i' - 1$) = choose(S')

{1} (choose(S') ∈ rest(S'))
{2} member(choose(S'), list_of_finite_set(rest(S')))
{3} $i' = 0$
{4} empty?(S')
{5} $i' = j''$

Hiding formulas: 1,
Using lemma nth_member[T],
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
This completes the proof of `unique_member_list_of_finite_set.2.2.2.1.2`.
`unique_member_list_of_finite_set.2.2.2.2:`

{-1} is_finite[T](S')
{-2} $i' < \text{length}(\text{list\_of\_finite\_set}(S'))$
{-3} $j'' < \text{length}(\text{list\_of\_finite\_set}(S'))$
{-4} $j' \geq 0$
{-5} $\forall$ (S: finite_set[T], i, j: below(length(list_of_finite_set(S)))):
$j' = \text{card}(S) \wedge \text{nth}(\text{list\_of\_finite\_set}(S), i) = \text{nth}(\text{list\_of\_finite\_set}(S), j)$
$\supset i = j$
{-6} $1 + j' = \text{card}(S')$
{-7} nth(cons(choose(S'), list_of_finite_set(rest(S'))), i') =
nth(cons(choose(S'), list_of_finite_set(rest(S'))), j'')

{1} $j'' = 0$
{2} $i' = 0$
{3} empty?(S')
{4} $i' = j''$

Expanding the definition of nth,

Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in -5 with the terms: rest(S'), $i' - 1$, $j'' - 1$,
Rewriting using card_rest, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unique_member_list_of_finite_set.2.2.2.2`.
Q.E.D.

C.96 Proofs for General (constants.pvs)

C.96.1 General.Byte_TCC1

Terse proof for `Byte_TCC1`.
`Byte_TCC1`:

{1} $0 < \text{max\_byte}$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Byte_TCC1`.
Q.E.D.

C.96.2 General.less_than_max_byte

Terse proof for `less_than_max_byte`.
`less_than_max_byte`:

{1} $\forall\ (n\colon \text{nat})\colon\ n < \text{expt}(2,\ 8) \supset n < \text{max\_byte}$

Repeatedly Skolemizing and flattening,
Expanding the definition of max_byte,
Applying bits_per_byte_minimum
Expanding the definition of min_bits_per_byte,
Applying both_sides_expt_gt1_le
Instantiating the top quantifier in -1 with the terms: 2, 8, bits_per_byte,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `less_than_max_byte`.
Q.E.D.

C.97 Proofs for Graph (graph.pvs)

C.97.1 Graph.path?_TCC1

Terse proof for `path?_TCC1`.
`path?_TCC1`:

{1} $\forall\ (g\colon \text{graph\_type},\ l\colon (\text{path\_list?}[T]))\colon$
$(\forall\ (i\colon \text{below}(\text{length}(l)))\colon\ g\text{`nodes}(\text{nth}(l,\ i))) \supset$
$(\forall\ (i_1\colon \text{below}(\text{length}[T](l) - 1))\colon\ i_1 < \text{length}[T](l))$

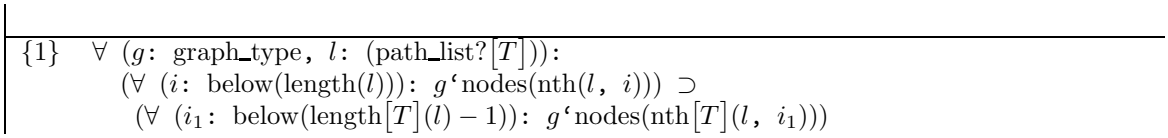
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path?_TCC1`.

Q.E.D.

C.97.2 Graph.path?_TCC2

Terse proof for `path?_TCC2`.

`path?_TCC2:`

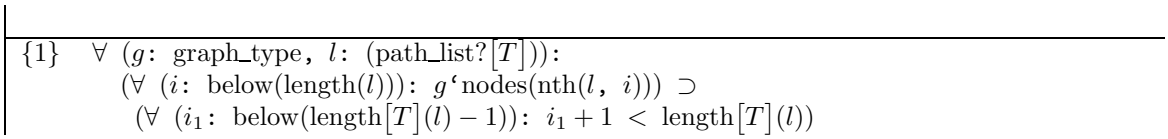


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path?_TCC2`.
Q.E.D.

C.97.3 Graph.path?_TCC3

Terse proof for `path?_TCC3`.

`path?_TCC3:`

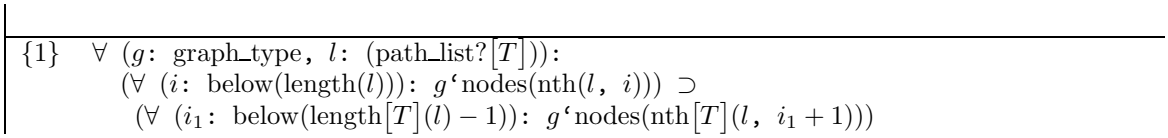


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path?_TCC3`.
Q.E.D.

C.97.4 Graph.path?_TCC4

Terse proof for `path?_TCC4`.

`path?_TCC4:`

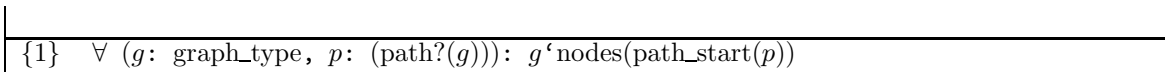


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path?_TCC4`.
Q.E.D.

C.97.5 Graph.nodes_path_start

Terse proof for `nodes_path_start`.

`nodes_path_start:`



Repeatedly Skolemizing and flattening,
Expanding the definition of path_list?,
Expanding the definition of path_start,

Expanding the definition of path?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -2 with the terms: 0,
we get 2 subgoals:
`nodes_path_start.1:`

{-1} length(p') $\geq$ 1
{-2} g'`nodes(nth(p', 0))
{-3} $\forall$ (i: below(length(p') $-$ 1)): g'`edges(nth(p', i), nth(p', 1 + i))

{1} g'`nodes(car(p'))

Expanding the definition of nth,
which is trivially true.
This completes the proof of `nodes_path_start.1`.
`nodes_path_start.2:`

{-1} length(p') $\geq$ 1
{-2} $\forall$ (i: below(length(p') $-$ 1)): g'`edges(nth(p', i), nth(p', 1 + i))

{1} 0 $<$ length$\lceil T \rceil$(p')
{2} g'`nodes(car(p'))

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nodes_path_start.2`.
Q.E.D.

C.97.6 Graph.nodes_path_end

Terse proof for `nodes_path_end`.
`nodes_path_end:`

{1} $\forall$ (g: graph_type, p: (path?(g))): g`nodes(path_end(p))

Repeatedly Skolemizing and flattening,
Expanding the definition of path_list?,
Expanding the definition of path_end,
Expanding the definition of path?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nodes_path_end`.
Q.E.D.

C.97.7 Graph.path_head_TCC1

Terse proof for `path_head_TCC1`.
`path_head_TCC1:`

{1} $\forall$ (g: graph_type, p: (path?(g)), n: upto(length$\lceil T \rceil$(p))):
$n \geq 1 \supset$ path_list?$\lceil T \rceil$(head$\lceil T \rceil$(p, n))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path_head_TCC1`.
Q.E.D.

C.97.8 Graph.path_head

Terse proof for `path_head`.

`path_head:`

```
{1}  ∀ (g: graph_type, p: (path?(g)), n: upto(length(p))):
       n ≥ 1 ⊃ path?(g)(head(p, n))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of path?,
Installing automatic rewrites from: length_head length_tail nth_head nth_tail
Applying propositional simplification,
we get 2 subgoals:

`path_head.1:`

```
{-1}  path_list?[T](p')
{-2}  ∀ (i: below(length(p'))): g'`nodes(nth(p', i))
{-3}  ∀ (i: below(length(p') − 1)): g'`edges(nth(p', i), nth(p', 1 + i))
{-4}  n' ≤ length(p')
{-5}  n' ≥ 1
{1}   ∀ (i: below(length(head(p', n')))): g'`nodes(nth(head(p', n'), i))
```

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
This completes the proof of `path_head.1`.

`path_head.2:`

```
{-1}  path_list?[T](p')
{-2}  ∀ (i: below(length(p'))): g'`nodes(nth(p', i))
{-3}  ∀ (i: below(length(p') − 1)): g'`edges(nth(p', i), nth(p', 1 + i))
{-4}  n' ≤ length(p')
{-5}  n' ≥ 1
{1}   ∀ (i: below(length(head(p', n')) − 1)):
        g'`edges(nth(head(p', n'), i), nth(head(p', n'), 1 + i))
```

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
This completes the proof of `path_head.2`.
Q.E.D.

C.97.9 Graph.path_tail_TCC1

Terse proof for `path_tail_TCC1`.

`path_tail_TCC1:`

```
{1}  ∀ (g: graph_type, p: (path?(g)), n: upto(length[T](p))):
       length(p) − n ≥ 1 ⊃ path_list?[T](tail[T](p, n))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of path_list?,
Rewriting using length_tail, matching in *,
This completes the proof of `path_tail_TCC1`.

Q.E.D.

C.97.10 Graph.path_tail

Terse proof for `path_tail`.

`path_tail`:

```
{1}  ∀ (g: graph_type, p: (path?(g)), n: upto(length(p))):
       length(p) − n ≥ 1 ⊃ path?(g)(tail(p, n))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of path?,
Installing automatic rewrites from: length_head length_tail nth_head nth_tail
Applying propositional simplification,
we get 2 subgoals:

`path_tail.1`:

```
{-1}  path_list?[T](p′)
{-2}  ∀ (i: below(length(p′))): g′‘nodes(nth(p′, i))
{-3}  ∀ (i: below(length(p′) − 1)): g′‘edges(nth(p′, i), nth(p′, 1 + i))
{-4}  n′ ≤ length(p′)
{-5}  length(p′) − n′ ≥ 1
{1}  ∀ (i: below(length(tail(p′, n′)))): g′‘nodes(nth(tail(p′, n′), i))
```

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
This completes the proof of `path_tail.1`.

`path_tail.2`:

```
{-1}  path_list?[T](p′)
{-2}  ∀ (i: below(length(p′))): g′‘nodes(nth(p′, i))
{-3}  ∀ (i: below(length(p′) − 1)): g′‘edges(nth(p′, i), nth(p′, 1 + i))
{-4}  n′ ≤ length(p′)
{-5}  length(p′) − n′ ≥ 1
{1}  ∀ (i: below(length(tail(p′, n′)) − 1)):
       g′‘edges(nth(tail(p′, n′), i), nth(tail(p′, n′), 1 + i))
```

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_tail.2`.
Q.E.D.

C.97.11 Graph.path_cdr_TCC1

Terse proof for `path_cdr_TCC1`.

`path_cdr_TCC1`:

```
{1}  ∀ (g: graph_type, p: (path?(g))): cons?[T](p)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path_cdr_TCC1`.

Q.E.D.

C.97.12 Graph.path_cdr

Terse proof for `path_cdr`.

`path_cdr:`

{1} $\forall$ (g: graph_type, p: (path?(g))):
path_list?$[T]$(cdr(p)) $\supset$ path?(g)(cdr(p))

Repeatedly Skolemizing and flattening,
Expanding the definition of path_list?,
Installing automatic rewrites from: length_cdr tail
Simplifying, rewriting, and recording with decision procedures,
Using lemma path_tail,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cdr`.
Q.E.D.

C.97.13 Graph.path_concat_path

Terse proof for `path_concat_path`.

`path_concat_path:`

{1} $\forall$ (g: graph_type, p_1, p_2: (path?(g))):
concatable?(p_1, p_2) $\supset$ path?(g)(concat_path(p_1, p_2))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_list? path? nth_append_left nth_append_right length_append length_cdr length_concat_path concat_path
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Applying propositional simplification,
we get 2 subgoals:

`path_concat_path.1:`

{-1} length(p'_1) $\geq$ 1
{-2} $\forall$ (i: below(length(p'_1))): g'`nodes(nth(p'_1, i))
{-3} $\forall$ (i: below(length(p'_1) $-$ 1)):
g'`edges(nth(p'_1, i), nth(p'_1, $1+i$))
{-4} length(p'_2) $\geq$ 1
{-5} $\forall$ (i: below(length(p'_2))): g'`nodes(nth(p'_2, i))
{-6} $\forall$ (i: below(length(p'_2) $-$ 1)):
g'`edges(nth(p'_2, i), nth(p'_2, $1+i$))
{-7} concatable?(p'_1, p'_2)

{1} $\forall$ (i: below(length(append(p'_1, cdr(p'_2))))):
g'`nodes(nth(append(p'_1, cdr(p'_2)), i))

Hiding formulas: -3, -6,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on i!1 < length(p1!1),
we get 2 subgoals:

`path_concat_path.1.1:`

{-1}	$i' < \text{length}(p'_1)$
{-2}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 1$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1)))\colon\ g\text{`}\text{nodes}(\text{nth}(p'_1,\ i))$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2)))\colon\ g\text{`}\text{nodes}(\text{nth}(p'_2,\ i))$
{-7}	$\text{concatable?}(p'_1,\ p'_2)$
{1}	$g\text{`}\text{nodes}(\text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ i'))$

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of `path_concat_path.1.1`.

`path_concat_path.1.2:`

{-1}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 1$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1)))\colon\ g\text{`}\text{nodes}(\text{nth}(p'_1,\ i))$
{-4}	$\text{length}(p'_2) \geq 1$
{-5}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2)))\colon\ g\text{`}\text{nodes}(\text{nth}(p'_2,\ i))$
{-6}	$\text{concatable?}(p'_1,\ p'_2)$
{1}	$i' < \text{length}(p'_1)$
{2}	$g\text{`}\text{nodes}(\text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ i'))$

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -5 with the terms: $i' - \text{length}(p'_1) + 1$,

Expanding the definition of nth,

which is trivially true.

This completes the proof of `path_concat_path.1.2`.

`path_concat_path.2:`

{-1}	$\text{length}(p'_1) \geq 1$
{-2}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1)))\colon\ g\text{`}\text{nodes}(\text{nth}(p'_1,\ i))$
{-3}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1) - 1))\colon$ $g\text{`}\text{edges}(\text{nth}(p'_1,\ i),\ \text{nth}(p'_1,\ 1 + i))$
{-4}	$\text{length}(p'_2) \geq 1$
{-5}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2)))\colon\ g\text{`}\text{nodes}(\text{nth}(p'_2,\ i))$
{-6}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2) - 1))\colon$ $g\text{`}\text{edges}(\text{nth}(p'_2,\ i),\ \text{nth}(p'_2,\ 1 + i))$
{-7}	$\text{concatable?}(p'_1,\ p'_2)$
{1}	$\forall\ (i\colon\ \text{below}(\text{length}(\text{append}(p'_1,\ \text{cdr}(p'_2))) - 1))\colon$ $g\text{`}\text{edges}$ $(\text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ i),\ \text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ 1 + i))$

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on 1 + i!1 < length(p1!1),

we get 2 subgoals:

`path_concat_path.2.1:`

{-1}	$1 + i' < \text{length}(p'_1)$
{-2}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1)))\colon\ g'\text{`nodes}(\text{nth}(p'_1,\ i))$
{-5}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1) - 1))\colon$ $g'\text{`edges}(\text{nth}(p'_1,\ i),\ \text{nth}(p'_1,\ 1 + i))$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2)))\colon\ g'\text{`nodes}(\text{nth}(p'_2,\ i))$
{-8}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2) - 1))\colon$ $g'\text{`edges}(\text{nth}(p'_2,\ i),\ \text{nth}(p'_2,\ 1 + i))$
{-9}	$\text{concatable?}(p'_1,\ p'_2)$
{1}	$g'\text{`edges}$ $(\text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ i'),\ \text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
This completes the proof of `path_concat_path.2.1`.
`path_concat_path.2.2:`

{-1}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1)))\colon\ g'\text{`nodes}(\text{nth}(p'_1,\ i))$
{-4}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1) - 1))\colon$ $g'\text{`edges}(\text{nth}(p'_1,\ i),\ \text{nth}(p'_1,\ 1 + i))$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2)))\colon\ g'\text{`nodes}(\text{nth}(p'_2,\ i))$
{-7}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2) - 1))\colon$ $g'\text{`edges}(\text{nth}(p'_2,\ i),\ \text{nth}(p'_2,\ 1 + i))$
{-8}	$\text{concatable?}(p'_1,\ p'_2)$
{1}	$1 + i' < \text{length}(p'_1)$
{2}	$g'\text{`edges}$ $(\text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ i'),\ \text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ 1 + i'))$

Case splitting on 1 + i!1 = length(p1!1),
we get 2 subgoals:
`path_concat_path.2.2.1:`

{-1}	$1 + i' = \text{length}(p'_1)$
{-2}	$i' < \text{length}(p'_1) + \text{length}(p'_2) - 2$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1)))\colon\ g'\text{`nodes}(\text{nth}(p'_1,\ i))$
{-5}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_1) - 1))\colon$ $g'\text{`edges}(\text{nth}(p'_1,\ i),\ \text{nth}(p'_1,\ 1 + i))$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2)))\colon\ g'\text{`nodes}(\text{nth}(p'_2,\ i))$
{-8}	$\forall\ (i\colon\ \text{below}(\text{length}(p'_2) - 1))\colon$ $g'\text{`edges}(\text{nth}(p'_2,\ i),\ \text{nth}(p'_2,\ 1 + i))$
{-9}	$\text{concatable?}(p'_1,\ p'_2)$
{1}	$1 + i' < \text{length}(p'_1)$
{2}	$g'\text{`edges}$ $(\text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ i'),\ \text{nth}(\text{append}(p'_1,\ \text{cdr}(p'_2)),\ 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of concatable?,

Expanding the definition of path_end,
Expanding the definition of path_start,
Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -9,
Instantiating the top quantifier in -8 with the terms: 0,
Expanding the definition of nth,
which is trivially true.
This completes the proof of `path_concat_path.2.2.1`.
`path_concat_path.2.2.2`:

$$
\begin{array}{ll}
\{\text{-}1\} & i' < \text{length}(p_1') + \text{length}(p_2') - 2 \\
\{\text{-}2\} & \text{length}(p_1') \geq 1 \\
\{\text{-}3\} & \forall\ (i\text{: below}(\text{length}(p_1'))):\ g\text{`nodes}(\text{nth}(p_1', i)) \\
\{\text{-}4\} & \forall\ (i\text{: below}(\text{length}(p_1') - 1)): \\
 & \quad g\text{`edges}(\text{nth}(p_1', i), \text{nth}(p_1', 1 + i)) \\
\{\text{-}5\} & \text{length}(p_2') \geq 1 \\
\{\text{-}6\} & \forall\ (i\text{: below}(\text{length}(p_2'))):\ g\text{`nodes}(\text{nth}(p_2', i)) \\
\{\text{-}7\} & \forall\ (i\text{: below}(\text{length}(p_2') - 1)): \\
 & \quad g\text{`edges}(\text{nth}(p_2', i), \text{nth}(p_2', 1 + i)) \\
\{\text{-}8\} & \text{concatable?}(p_1', p_2') \\
\hline
\{1\} & 1 + i' = \text{length}(p_1') \\
\{2\} & 1 + i' < \text{length}(p_1') \\
\{3\} & g\text{`edges} \\
 & \quad (\text{nth}(\text{append}(p_1', \text{cdr}(p_2')), i'), \text{nth}(\text{append}(p_1', \text{cdr}(p_2')), 1 + i'))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in -7 with the terms: $i' - \text{length}(p_1') + 1$,
Expanding the definition of nth,
which is trivially true.
This completes the proof of `path_concat_path.2.2.2`.
Q.E.D.

C.97.14 Graph.trans_closure_char_graph

Terse proof for `trans_closure_char_graph`.
`trans_closure_char_graph`:

$$
\begin{array}{ll}
\{1\} & \forall\ (g\text{: graph\_type}, x, y\text{: } (g\text{`nodes})): \\
 & \quad \text{transitive\_closure}(g\text{`edges})(x, y) \equiv \\
 & \quad (\exists\ (l\text{: } (\text{path?}(g))): \\
 & \qquad \text{length}(l) \geq 2 \wedge \text{path\_start}(l) = x \wedge \text{path\_end}(l) = y)
\end{array}
$$

Repeatedly Skolemizing and flattening,
Using lemma trans_closure_char,
Installing automatic rewrites from: length_stable nth_stable
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`trans_closure_char_graph.1:`

{-1} transitive_closure(g`nodes)(x', y')
{-2} $\exists$ (l: list[(g`nodes)]):
length(l) $\geq$ 2 $\wedge$
path_start(l) = x' $\wedge$
path_end(l) = y' $\wedge$
($\forall$ (i: below(length(l) $-$ 1)):
g`edges(nth(l, i), nth(l, 1 + i)))
{-3} g`nodes(x')
{-4} g`nodes(y')

{1} $\exists$ (l: (path?(g'))): length(l) $\geq$ 2 $\wedge$ path_start(l) = x' $\wedge$ path_end(l) = y'

Keeping (-2 1) and hiding *,
Repeatedly Skolemizing and flattening,
Using lemma every_nth[T],
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: path? path_list?
Instantiating quantified variables,
we get 2 subgoals:
`trans_closure_char_graph.1.1:`

{-1} $\forall$ (i: below(length(l'))): g`nodes(nth(l', i))
{-2} every(g`nodes)(l')
{-3} length[T](l') $\geq$ 2
{-4} path_start(l') = x'
{-5} path_end(l') = y'
{-6} $\forall$ (i: below(length(l') $-$ 1)): g`edges(nth(l', i), nth(l', 1 + i))

{1} length(l') $\geq$ 2 $\wedge$ path_start(l') = x' $\wedge$ path_end(l') = y'

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of path_start,
Expanding the definition of path_end,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char_graph.1.1`.
`trans_closure_char_graph.1.2:`

{-1} $\forall$ (i: below(length(l'))): g`nodes(nth(l', i))
{-2} every(g`nodes)(l')
{-3} length[T](l') $\geq$ 2
{-4} path_start(l') = x'
{-5} path_end(l') = y'
{-6} $\forall$ (i: below(length(l') $-$ 1)): g`edges(nth(l', i), nth(l', 1 + i))

{1} ($\forall$ (i: below(length(l'))): g`nodes(nth(l', i))) $\wedge$
($\forall$ (i: below(length(l') $-$ 1)):
g`edges(nth(l', i), nth(l', 1 + i)))

Applying propositional simplification,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -7 with the terms: i',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char_graph.1.2`.

`trans_closure_char_graph.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & g'\text{`nodes}(x') \\
\{\text{-}2\} & g'\text{`nodes}(y') \\
\{\text{-}3\} & \exists\ (l\colon\ (\text{path?}(g'))) \colon\ \text{length}(l) \geq 2 \wedge \text{path\_start}(l) = x' \wedge \text{path\_end}(l) = y' \\
\hline
\{1\} & \text{transitive\_closure}(g'\text{`edges})(x',\ y') \\
\{2\} & \exists\ (l\colon\ \text{list}\big[(g'\text{`nodes})\big]) \colon \\
 & \quad \text{length}(l) \geq 2 \wedge \\
 & \quad\ \text{path\_start}(l) = x' \wedge \\
 & \quad\quad \text{path\_end}(l) = y' \wedge \\
 & \quad\quad\ (\forall\ (i\colon\ \text{below}(\text{length}(l) - 1)) \colon \\
 & \quad\quad\quad\ g'\text{`edges}(\text{nth}(l,\ i),\ \text{nth}(l,\ 1 + i)))
\end{array}
$$

Keeping (-3 2) and hiding *,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path? path_list?
Simplifying, rewriting, and recording with decision procedures,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of path_start,
Expanding the definition of path_end,
Using lemma every_nth$\big[T\big]$,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -5 with the terms: i',
we get 2 subgoals:
`trans_closure_char_graph.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & i' < \text{length}(l') - 1 \\
\{\text{-}2\} & \text{every}(g'\text{`nodes})(l') \\
\{\text{-}3\} & \forall\ (i\colon\ \text{below}(\text{length}(l'))) \colon\ g'\text{`nodes}(\text{nth}(l',\ i)) \\
\{\text{-}4\} & \text{length}(l') \geq 1 \\
\{\text{-}5\} & g'\text{`edges}(\text{nth}(l',\ i'),\ \text{nth}(l',\ 1 + i')) \\
\{\text{-}6\} & \text{car}(l') = x' \\
\{\text{-}7\} & \text{length}(l') \geq 2 \\
\{\text{-}8\} & \text{nth}(l',\ \text{length}(l') - 1) = y' \\
\hline
\{1\} & g'\text{`edges}(\text{nth}(l',\ i'),\ \text{nth}(l',\ 1 + i'))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char_graph.2.1`.
`trans_closure_char_graph.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & i' < \text{length}(l') - 1 \\
\{\text{-}2\} & \text{every}(g'\text{`nodes})(l') \\
\{\text{-}3\} & \forall\ (i\colon\ \text{below}(\text{length}(l'))) \colon\ g'\text{`nodes}(\text{nth}(l',\ i)) \\
\{\text{-}4\} & \text{length}(l') \geq 1 \\
\{\text{-}5\} & \text{car}(l') = x' \\
\{\text{-}6\} & \text{length}(l') \geq 2 \\
\{\text{-}7\} & \text{nth}(l',\ \text{length}(l') - 1) = y' \\
\hline
\{1\} & i' < \text{length}\big[T\big](l') - 1 \\
\{2\} & g'\text{`edges}(\text{nth}(l',\ i'),\ \text{nth}(l',\ 1 + i'))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char_graph.2.2`.

Q.E.D.

C.97.15 Graph.path_cycle_free

Terse proof for `path_cycle_free`.

`path_cycle_free:`

{1} $\forall$ (g: graph_type):
cycle_free?(g) $\equiv$
($\forall$ (l: (path?(g)), i_1, i_2: below(length(l))):
nth(l, i_1) = nth(l, i_2) $\supset$ i_1 = i_2)

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: car_tail car_head nth_tail nth_head length_head length_tail path_list? path_head path_tail path_start path_end

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`path_cycle_free.1:`

{-1} cycle_free?(g')

{1} $\forall$ (l: (path?(g')), i_1, i_2: below(length(l))):
nth(l, i_1) = nth(l, i_2) $\supset$ i_1 = i_2

Repeatedly Skolemizing and flattening,

Expanding the definition of cycle_free?,

Instantiating the top quantifier in -5 with the terms: nth(l', i'_1),

Applying propositional simplification,

we get 2 subgoals:

`path_cycle_free.1.1:`

{-1} path_list?$[T]$(l')
{-2} path?(g')(l')
{-3} i'_1 < length(l')
{-4} i'_2 < length(l')
{-5} nth(l', i'_1) = nth(l', i'_2)

{1} transitive_closure(g`edges)(nth(l', i'_1), nth(l', i'_1))
{2} i'_1 = i'_2

Rewriting using trans_closure_char_graph, matching in *,

Case splitting on i1!1 < i2!1,

we get 2 subgoals:

`path_cycle_free.1.1.1:`

{-1} i'_1 < i'_2
{-2} path_list?$[T]$(l')
{-3} path?(g')(l')
{-4} i'_1 < length(l')
{-5} i'_2 < length(l')
{-6} nth(l', i'_1) = nth(l', i'_2)

{1} $\exists$ (l: (path?(g'))):
length(l) $\geq$ 2 $\wedge$
car(l) = nth(l', i'_1) $\wedge$ nth(l, length(l) $-$ 1) = nth(l', i'_1)
{2} i'_1 = i'_2

Instantiating the top quantifier in 1 with the terms: tail(head(l', $i'_2 + 1$), i'_1),

we get 3 subgoals:

`path_cycle_free.1.1.1.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & i'_1 < i'_2 \\
\{\text{-}2\} & \text{path\_list?}[T](l') \\
\{\text{-}3\} & \text{path?}(g')(l') \\
\{\text{-}4\} & i'_1 < \text{length}(l') \\
\{\text{-}5\} & i'_2 < \text{length}(l') \\
\{\text{-}6\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & \text{length}(\text{tail}(\text{head}(l',\ i'_2+1),\ i'_1)) \geq 2 \wedge \\
 & \quad \text{car}(\text{tail}(\text{head}(l',\ i'_2+1),\ i'_1)) = \text{nth}(l',\ i'_1) \wedge \\
 & \quad\ \text{nth}(\text{tail}(\text{head}(l',\ i'_2+1),\ i'_1), \\
 & \qquad\quad \text{length}(\text{tail}(\text{head}(l',\ i'_2+1),\ i'_1)) - 1) \\
 & \quad\ = \text{nth}(l',\ i'_1) \\
\{2\} & i'_1 = i'_2
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.1.1`.

`path_cycle_free.1.1.1.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & i'_1 < i'_2 \\
\{\text{-}2\} & \text{path\_list?}[T](l') \\
\{\text{-}3\} & \text{path?}(g')(l') \\
\{\text{-}4\} & i'_1 < \text{length}(l') \\
\{\text{-}5\} & i'_2 < \text{length}(l') \\
\{\text{-}6\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & \text{length}(\text{tail}[T](\text{head}[T](l',\ 1+i'_2),\ i'_1)) \geq 1 \wedge \\
 & \quad \text{path?}(g')(\text{tail}[T](\text{head}[T](l',\ 1+i'_2),\ i'_1)) \\
\{2\} & i'_1 = i'_2
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.1.2`.

`path_cycle_free.1.1.1.3:`

$$
\begin{array}{ll}
\{\text{-}1\} & i'_1 < i'_2 \\
\{\text{-}2\} & \text{path\_list?}[T](l') \\
\{\text{-}3\} & \text{path?}(g')(l') \\
\{\text{-}4\} & i'_1 < \text{length}(l') \\
\{\text{-}5\} & i'_2 < \text{length}(l') \\
\{\text{-}6\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & i'_1 \leq 1 + i'_2 \\
\{2\} & i'_1 = i'_2
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_cycle_free.1.1.1.3`.

`path_cycle_free.1.1.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{path\_list?}[T](l') \\
\{\text{-}2\} & \text{path?}(g')(l') \\
\{\text{-}3\} & i'_1 < \text{length}(l') \\
\{\text{-}4\} & i'_2 < \text{length}(l') \\
\{\text{-}5\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & i'_1 < i'_2 \\
\{2\} & \exists\ (l\colon\ (\text{path?}(g'))): \\
 & \quad \text{length}(l) \geq 2 \wedge \\
 & \quad\ \text{car}(l) = \text{nth}(l',\ i'_1) \wedge \text{nth}(l,\ \text{length}(l) - 1) = \text{nth}(l',\ i'_1) \\
\{3\} & i'_1 = i'_2
\end{array}
$$

Instantiating the top quantifier in 2 with the terms: tail(head(l', $i'_1 + 1$), i'_2),
we get 3 subgoals:
`path_cycle_free.1.1.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{path\_list?}[T](l') \\
\{\text{-}2\} & \text{path?}(g')(l') \\
\{\text{-}3\} & i'_1 < \text{length}(l') \\
\{\text{-}4\} & i'_2 < \text{length}(l') \\
\{\text{-}5\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & i'_1 < i'_2 \\
\{2\} & \text{length}(\text{tail}(\text{head}(l',\ i'_1 + 1),\ i'_2)) \geq 2 \wedge \\
 & \quad \text{car}(\text{tail}(\text{head}(l',\ i'_1 + 1),\ i'_2)) = \text{nth}(l',\ i'_1) \wedge \\
 & \quad\ \text{nth}(\text{tail}(\text{head}(l',\ i'_1 + 1),\ i'_2), \\
 & \qquad\qquad \text{length}(\text{tail}(\text{head}(l',\ i'_1 + 1),\ i'_2)) - 1) \\
 & \quad\ = \text{nth}(l',\ i'_1) \\
\{3\} & i'_1 = i'_2
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cycle_free.1.1.2.1`.
`path_cycle_free.1.1.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{path\_list?}[T](l') \\
\{\text{-}2\} & \text{path?}(g')(l') \\
\{\text{-}3\} & i'_1 < \text{length}(l') \\
\{\text{-}4\} & i'_2 < \text{length}(l') \\
\{\text{-}5\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & \text{length}(\text{tail}[T](\text{head}[T](l',\ 1 + i'_1),\ i'_2)) \geq 1 \wedge \\
 & \quad \text{path?}(g')(\text{tail}[T](\text{head}[T](l',\ 1 + i'_1),\ i'_2)) \\
\{2\} & i'_1 < i'_2 \\
\{3\} & i'_1 = i'_2
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cycle_free.1.1.2.2`.
`path_cycle_free.1.1.2.3:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{path\_list?}[T](l') \\
\{\text{-}2\} & \text{path?}(g')(l') \\
\{\text{-}3\} & i'_1 < \text{length}(l') \\
\{\text{-}4\} & i'_2 < \text{length}(l') \\
\{\text{-}5\} & \text{nth}(l',\ i'_1) = \text{nth}(l',\ i'_2) \\
\hline
\{1\} & i'_2 \leq 1 + i'_1 \\
\{2\} & i'_1 < i'_2 \\
\{3\} & i'_1 = i'_2
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cycle_free.1.1.2.3`.
`path_cycle_free.1.2:`

{-1}	path_list?$[T]$(l')
{-2}	path?(g')(l')
{-3}	$i'_1 < \text{length}(l')$
{-4}	$i'_2 < \text{length}(l')$
{-5}	$\text{nth}(l', i'_1) = \text{nth}(l', i'_2)$
{1}	g'`nodes(nth(l', i'_1))
{2}	$i'_1 = i'_2$

Expanding the definition of path?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `path_cycle_free.1.2`.
`path_cycle_free.2:`

{-1}	$\forall\ (l\text{: (path?}(g'))\text{, } i_1\text{, } i_2\text{: below(length}(l))\text{):}$ $\text{nth}(l, i_1) = \text{nth}(l, i_2) \supset i_1 = i_2$
{1}	cycle_free?(g')

Expanding the definition of cycle_free?,
Repeatedly Skolemizing and flattening,
Rewriting using trans_closure_char_graph, matching in *,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -7 with the terms: l', 0, length(l') − 1,
we get 3 subgoals:
`path_cycle_free.2.1:`

{-1}	path_list?$[T]$(l')
{-2}	path?(g')(l')
{-3}	g'`nodes($x'$)`
{-4}	$\text{length}(l') \geq 2$
{-5}	$\text{car}(l') = x'$
{-6}	$\text{nth}(l', \text{length}(l') - 1) = x'$
{-7}	$\text{nth}(l', 0) = \text{nth}(l', \text{length}(l') - 1) \supset 0 = \text{length}(l') - 1$

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of nth,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cycle_free.2.1`.
`path_cycle_free.2.2:`

{-1}	path_list?$[T]$(l')
{-2}	path?(g')(l')
{-3}	g'`nodes($x'$)`
{-4}	$\text{length}(l') \geq 2$
{-5}	$\text{car}(l') = x'$
{-6}	$\text{nth}(l', \text{length}(l') - 1) = x'$
{1}	length$[T]$(l') − 1 ≥ 0

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cycle_free.2.2`.

`path_cycle_free.2.3:`

{-1}	path_list?$[T]$(l')
{-2}	path?(g')(l')
{-3}	g'`nodes(x')
{-4}	length(l') $\geq$ 2
{-5}	car(l') = x'
{-6}	nth(l', length(l') $-$ 1) = x'
{1}	0 $<$ length$[T]$(l')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_cycle_free.2.3`.
Q.E.D.

C.97.16 Graph.different_nodes_on_edge

Terse proof for `different_nodes_on_edge`.
`different_nodes_on_edge:`

{1}	$\forall$ (g: graph_type, x, y: (g`nodes)): cycle_free?($g$) $\wedge$ $g$`edges(x, y) $\supset$ $\neg$ x = y

Repeatedly Skolemizing and flattening,
Using lemma path_cycle_free,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: nth length path_list? path?
Instantiating the top quantifier in -1 with the terms: (:x', y':), 0, 1,
we get 2 subgoals:
`different_nodes_on_edge.1:`

{-1}	nth((:x', y':), 0) = nth((:x', y':), 1) $\supset$ 0 = 1
{-2}	g'`nodes(x')
{-3}	g'`nodes(y')
{-4}	cycle_free?(g')
{-5}	g'`edges(x', y')
{-6}	x' = y'

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.1`.
`different_nodes_on_edge.2:`

{-1}	g'`nodes(x')
{-2}	g'`nodes(y')
{-3}	cycle_free?(g')
{-4}	g'`edges(x', y')
{-5}	x' = y'
{1}	($\forall$ (i: below(length((:x', y':)))): g'`nodes(nth((:$x'$, $y'$:), $i$))) $\wedge$ ($\forall$ ($i$: below(length((:$x'$, $y'$:)) $-$ 1)): $g'$`edges(nth((:x', y':), i), nth((:y':), i)))

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
Case splitting on i!1 = 0,
we get 2 subgoals:

`different_nodes_on_edge.2.1:`

{-1}	$i' = 0$
{-2}	$i' < 2$
{-3}	g'`nodes`(y')
{-4}	cycle_free?(g')
{-5}	g'`edges`(y', y')
{-6}	$x' = y'$
{1}	g'`nodes`$(\text{nth}((:y', y':), i'))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.2.1`.
`different_nodes_on_edge.2.2:`

{-1}	$i' < 2$
{-2}	g'`nodes`(y')
{-3}	cycle_free?(g')
{-4}	g'`edges`(y', y')
{-5}	$x' = y'$
{1}	$i' = 0$
{2}	g'`nodes`$(\text{nth}((:y', y':), i'))$

Case splitting on i!1 = 1,
we get 2 subgoals:
`different_nodes_on_edge.2.2.1:`

{-1}	$i' = 1$
{-2}	$i' < 2$
{-3}	g'`nodes`(y')
{-4}	cycle_free?(g')
{-5}	g'`edges`(y', y')
{-6}	$x' = y'$
{1}	$i' = 0$
{2}	g'`nodes`$(\text{nth}((:y', y':), i'))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.2.2.1`.
`different_nodes_on_edge.2.2.2:`

{-1}	$i' < 2$
{-2}	g'`nodes`(y')
{-3}	cycle_free?(g')
{-4}	g'`edges`(y', y')
{-5}	$x' = y'$
{1}	$i' = 1$
{2}	$i' = 0$
{3}	g'`nodes`$(\text{nth}((:y', y':), i'))$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `different_nodes_on_edge.2.2.2`.
Q.E.D.

C.97.17 Graph.tree_path

Terse proof for `tree_path`.

`tree_path:`

{1} ∀ (g: graph_type, p_1, p_2: (path?(g))):
tree?(g) ∧ path_end(p_1) = path_end(p_2) ∧ length(p_1) = length(p_2) ⊃ p_1 = p_2

For the top quantifier in 1, we introduce Skolem constants: (g' _ _),
Inducting on p1 on formula 1 using induction scheme path_list_induction$[T]$,
we get 3 subgoals:
`tree_path.1:`

{1} path?(g')(p_1')
{2} ∀ (p_2: (path?(g'))):
tree?(g') ∧ path_end(p_1') = path_end(p_2) ∧ length(p_1') = length(p_2) ⊃ p_1' = p_2

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.1`.
`tree_path.2:`

{1} ∀ (t: T):
path?(g')(cons(t, null)) ⊃
(∀ (p_2: (path?(g'))):
tree?(g') ∧
path_end(cons(t, null)) = path_end(p_2) ∧
length(cons(t, null)) = length(p_2)
⊃ cons(t, null) = p_2)

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: length path_end nth
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
Applying extensionality,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `tree_path.2`.
`tree_path.3:`

{1} ∀ (t: T, tail: (path_list?)):
(path?(g')(tail) ⊃
(∀ (p_2: (path?(g'))):
tree?(g') ∧ path_end(tail) = path_end(p_2) ∧ length(tail) = length(p_2) ⊃
tail = p_2))
⊃
path?(g')(cons(t, tail)) ⊃
(∀ (p_2: (path?(g'))):
tree?(g') ∧
path_end(cons(t, tail)) = path_end(p_2) ∧
length(cons(t, tail)) = length(p_2)
⊃ cons(t, tail) = p_2)

Repeatedly Skolemizing and flattening,
Using lemma path_cdr,
Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -5 with the terms: $\mathrm{cdr}(p_2')$,

we get 2 subgoals:

`tree_path.3.1:`

{-1}	$\mathrm{path?}(g')(\mathrm{tail}')$
{-2}	$\mathrm{path\_list?}[T](p_2')$
{-3}	$\mathrm{path?}(g')(p_2')$
{-4}	$\mathrm{path\_list?}(\mathrm{tail}')$
{-5}	$\mathrm{path\_end}(\mathrm{tail}') = \mathrm{path\_end}(\mathrm{cdr}(p_2')) \wedge \mathrm{length}(\mathrm{tail}') = \mathrm{length}(\mathrm{cdr}(p_2')) \supset \mathrm{tail}' = \mathrm{cdr}(p_2')$
{-6}	$\mathrm{path?}(g')(\mathrm{cons}(t', \mathrm{tail}'))$
{-7}	$\mathrm{tree?}(g')$
{-8}	$\mathrm{path\_end}(\mathrm{cons}(t', \mathrm{tail}')) = \mathrm{path\_end}(p_2')$
{-9}	$\mathrm{length}(\mathrm{tail}') = \mathrm{length}(\mathrm{cdr}(p_2'))$
{1}	$\mathrm{null?}(p_2')$
{2}	$\mathrm{cons}(t', \mathrm{tail}') = p_2'$

Expanding the definition of path_list?,

Applying propositional simplification,

we get 2 subgoals:

`tree_path.3.1.1:`

{-1}	$\mathrm{tail}' = \mathrm{cdr}(p_2')$
{-2}	$\mathrm{path?}(g')(\mathrm{tail}')$
{-3}	$\mathrm{length}(p_2') \geq 1$
{-4}	$\mathrm{path?}(g')(p_2')$
{-5}	$\mathrm{length}(\mathrm{tail}') \geq 1$
{-6}	$\mathrm{path?}(g')(\mathrm{cons}(t', \mathrm{tail}'))$
{-7}	$\mathrm{tree?}(g')$
{-8}	$\mathrm{path\_end}(\mathrm{cons}(t', \mathrm{tail}')) = \mathrm{path\_end}(p_2')$
{-9}	$\mathrm{length}(\mathrm{tail}') = \mathrm{length}(\mathrm{cdr}(p_2'))$
{1}	$\mathrm{null?}(p_2')$
{2}	$\mathrm{cons}(t', \mathrm{tail}') = p_2'$

Applying extensionality,

Hiding formulas: -2,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -4 with the terms: 0,

we get 2 subgoals:

`tree_path.3.1.1.1:`

{-1}	$\text{tail}' = \text{cdr}(p'_2)$
{-2}	$\text{length}(p'_2) \geq 1$
{-3}	$\forall\ (i\colon \text{below}(\text{length}(p'_2)))\colon g'\text{`nodes}(\text{nth}(p'_2,\ i))$
{-4}	$g'\text{`edges}(\text{nth}(p'_2,\ 0),\ \text{nth}(p'_2,\ 1+0))$
{-5}	$\text{length}(\text{tail}') \geq 1$
{-6}	$\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t',\ \text{tail}'))))\colon g'\text{`nodes}(\text{nth}(\text{cons}(t',\ \text{tail}'),\ i))$
{-7}	$\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t',\ \text{tail}')) - 1))\colon$ $g'\text{`edges}(\text{nth}(\text{cons}(t',\ \text{tail}'),\ i),\ \text{nth}(\text{cons}(t',\ \text{tail}'),\ 1+i))$
{-8}	$\text{tree?}(g')$
{-9}	$\text{path\_end}(\text{cons}(t',\ \text{tail}')) = \text{path\_end}(p'_2)$
{-10}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$t' = \text{car}(p'_2)$
{2}	$\text{null?}(p'_2)$

Instantiating the top quantifier in -7 with the terms: 0,
we get 2 subgoals:
`tree_path.3.1.1.1.1:`

{-1}	$\text{tail}' = \text{cdr}(p'_2)$
{-2}	$\text{length}(p'_2) \geq 1$
{-3}	$\forall\ (i\colon \text{below}(\text{length}(p'_2)))\colon g'\text{`nodes}(\text{nth}(p'_2,\ i))$
{-4}	$g'\text{`edges}(\text{nth}(p'_2,\ 0),\ \text{nth}(p'_2,\ 1+0))$
{-5}	$\text{length}(\text{tail}') \geq 1$
{-6}	$\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t',\ \text{tail}'))))\colon g'\text{`nodes}(\text{nth}(\text{cons}(t',\ \text{tail}'),\ i))$
{-7}	$g'\text{`edges}(\text{nth}(\text{cons}(t',\ \text{tail}'),\ 0),\ \text{nth}(\text{cons}(t',\ \text{tail}'),\ 1+0))$
{-8}	$\text{tree?}(g')$
{-9}	$\text{path\_end}(\text{cons}(t',\ \text{tail}')) = \text{path\_end}(p'_2)$
{-10}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$t' = \text{car}(p'_2)$
{2}	$\text{null?}(p'_2)$

Installing automatic rewrites from: nth
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of tree?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -9 with the terms: t', $\text{car}(p'_2)$, $\text{car}(\text{tail}')$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.1.1.1`.
`tree_path.3.1.1.1.2:`

{-1}	$\text{tail}' = \text{cdr}(p'_2)$
{-2}	$\text{length}(p'_2) \geq 1$
{-3}	$\forall\ (i\colon \text{below}(\text{length}(p'_2)))\colon g'\text{`nodes}(\text{nth}(p'_2,\ i))$
{-4}	$g'\text{`edges}(\text{nth}(p'_2,\ 0),\ \text{nth}(p'_2,\ 1+0))$
{-5}	$\text{length}(\text{tail}') \geq 1$
{-6}	$\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t',\ \text{tail}'))))\colon g'\text{`nodes}(\text{nth}(\text{cons}(t',\ \text{tail}'),\ i))$
{-7}	$\text{tree?}(g')$
{-8}	$\text{path\_end}(\text{cons}(t',\ \text{tail}')) = \text{path\_end}(p'_2)$
{-9}	$\text{length}(\text{tail}') = \text{length}(\text{cdr}(p'_2))$
{1}	$0 < \text{length}\lfloor T\rceil(\text{cons}\lfloor T\rceil(t',\ \text{tail}')) - 1$
{2}	$t' = \text{car}(p'_2)$
{3}	$\text{null?}(p'_2)$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.1.1.2`.
`tree_path.3.1.1.2:`

{-1} $\text{tail}' = \text{cdr}(p_2')$
{-2} $\text{length}(p_2') \geq 1$
{-3} $\forall\ (i\colon \text{below}(\text{length}(p_2'))) \colon g'\text{`nodes}(\text{nth}(p_2', i))$
{-4} $\text{length}(\text{tail}') \geq 1$
{-5} $\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t', \text{tail}')))) \colon g'\text{`nodes}(\text{nth}(\text{cons}(t', \text{tail}'), i))$
{-6} $\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t', \text{tail}')) - 1)) \colon$
$g'\text{`edges}(\text{nth}(\text{cons}(t', \text{tail}'), i), \text{nth}(\text{cons}(t', \text{tail}'), 1 + i))$
{-7} $\text{tree?}(g')$
{-8} $\text{path\_end}(\text{cons}(t', \text{tail}')) = \text{path\_end}(p_2')$
{-9} $\text{length}(\text{tail}') = \text{length}(\text{cdr}(p_2'))$

{1} $0 < \text{length}\big[T\big](p_2') - 1$
{2} $t' = \text{car}(p_2')$
{3} $\text{null?}(p_2')$

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.1.2`.
`tree_path.3.1.2:`

{-1} $\text{path?}(g')(\text{tail}')$
{-2} $\text{length}(p_2') \geq 1$
{-3} $\text{path?}(g')(p_2')$
{-4} $\text{length}(\text{tail}') \geq 1$
{-5} $\text{path?}(g')(\text{cons}(t', \text{tail}'))$
{-6} $\text{tree?}(g')$
{-7} $\text{path\_end}(\text{cons}(t', \text{tail}')) = \text{path\_end}(p_2')$
{-8} $\text{length}(\text{tail}') = \text{length}(\text{cdr}(p_2'))$

{1} $\text{path\_end}(\text{tail}') = \text{path\_end}(\text{cdr}(p_2'))$
{2} $\text{null?}(p_2')$
{3} $\text{cons}(t', \text{tail}') = p_2'$

Installing automatic rewrites from: path_end nth length
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path.3.1.2`.
`tree_path.3.2:`

{-1} $\text{path?}(g')(\text{tail}')$
{-2} $\text{path\_list?}\big[T\big](p_2')$
{-3} $\text{path?}(g')(p_2')$
{-4} $\text{path\_list?}(\text{tail}')$
{-5} $\text{path?}(g')(\text{cons}(t', \text{tail}'))$
{-6} $\text{tree?}(g')$
{-7} $\text{path\_end}(\text{cons}(t', \text{tail}')) = \text{path\_end}(p_2')$
{-8} $\text{length}(\text{tail}') = \text{length}(\text{cdr}(p_2'))$

{1} $\text{path\_list?}\big[T\big](\text{cdr}\big[T\big](p_2')) \wedge \text{path?}(g')(\text{cdr}\big[T\big](p_2'))$
{2} $\text{null?}(p_2')$
{3} $\text{cons}(t', \text{tail}') = p_2'$

Using lemma path_cdr,
Expanding the definition of path_list?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `tree_path.3.2`.
Q.E.D.

C.97.18 Graph.root_path_TCC1

Terse proof for `root_path_TCC1`.
`root_path_TCC1:`

```
{1}  ∀ (g: graph_type, n: (g`nodes), p: (path?(g))): (g`nodes)(path_start[T](p))
```

Repeatedly Skolemizing and flattening,
Rewriting using nodes_path_start, matching in *,
This completes the proof of `root_path_TCC1`.
Q.E.D.

C.97.19 Graph.path_length_bound_TCC1

Terse proof for `path_length_bound_TCC1`.
`path_length_bound_TCC1:`

```
{1}  ∀ (g: graph_type):
       cycle_free?(g) ∧ finite?(g) ⊃ (∀ (p: (path?(g))): is_finite[T](g`nodes))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of finite?,
which is trivially true.
This completes the proof of `path_length_bound_TCC1`.
Q.E.D.

C.97.20 Graph.path_length_bound

Terse proof for `path_length_bound`.
`path_length_bound:`

```
{1}  ∀ (g: graph_type):
       finite?(g) ∧ cycle_free?(g) ⊃
        (∀ (p: (path?(g))): length(p) ≤ card(g`nodes))
```

Repeatedly Skolemizing and flattening,
Using lemma list_pred_card[T],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`path_length_bound.1:`

```
{-1}  ∃ (n1, n2: below(length(p′))): ¬ n1 = n2 ∧ nth(p′, n1) = nth(p′, n2)
{-2}  path_list?[T](p′)
{-3}  path?(g′)(p′)
{-4}  finite?(g′)
{-5}  cycle_free?(g′)
{1}   length(p′) ≤ card(g′`nodes)
```

Repeatedly Skolemizing and flattening,
Using lemma path_cycle_free,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_length_bound.1`.
`path_length_bound.2:`

{-1}	path_list?$[T]$(p')
{-2}	path?(g')(p')
{-3}	finite?(g')
{-4}	cycle_free?(g')
{1}	every(g'`nodes)(p')
{2}	length(p') $\leq$ card(g'`nodes)

Rewriting using every_nth, matching in *,
Expanding the definition of path?,
Applying disjunctive simplification to flatten sequent,
This completes the proof of `path_length_bound.2`.
`path_length_bound.3:`

{-1}	path_list?$[T]$(p')
{-2}	path?(g')(p')
{-3}	finite?(g')
{-4}	cycle_free?(g')
{1}	is_finite(g'`nodes)
{2}	length(p') $\leq$ card(g'`nodes)

Expanding the definition of finite?,
which is trivially true.
This completes the proof of `path_length_bound.3`.
Q.E.D.

C.97.21 Graph.non_root_path

Terse proof for `non_root_path`.
`non_root_path:`

{1}	$\forall$ (g: graph_type, n: (g`nodes)): roots(g)(n) $\vee$ ($\exists$ (p: (path?(g))): length(p) = 2 $\wedge$ path_end(p) = n)

Repeatedly Skolemizing and flattening,
Expanding the definition of roots,
Repeatedly Skolemizing and flattening,
Case splitting on length[T]((: x!1, n!1 :)) = 2,
we get 2 subgoals:
`non_root_path.1:`

{-1}	length$[T]$((:x', n':)) = 2
{-2}	g'`nodes(x')
{-3}	g'`nodes(n')
{-4}	g'`edges(x', n')
{1}	$\exists$ (p: (path?(g'))): length(p) = 2 $\wedge$ path_end(p) = n'

Instantiating the top quantifier in 1 with the terms: (:x', n':),

we get 2 subgoals:

`non_root_path.1.1:`

{-1}	length$\lceil T \rceil$((:x', n':)) = 2
{-2}	g`nodes(x')
{-3}	g`nodes(n')
{-4}	g`edges(x', n')
{1}	length((:x', n':)) = 2 ∧ path_end((:x', n':)) = n'

Simplifying, rewriting, and recording with decision procedures,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `non_root_path.1.1`.

`non_root_path.1.2:`

{-1}	length$\lceil T \rceil$((:x', n':)) = 2
{-2}	g`nodes(x')
{-3}	g`nodes(n')
{-4}	g`edges(x', n')
{1}	path_list?$\lceil T \rceil$((:x', n':)) ∧ path?(g')((:x', n':))

Expanding the definition of path_list?,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of path?,

Applying propositional simplification,

we get 2 subgoals:

`non_root_path.1.2.1:`

{-1}	length$\lceil T \rceil$((:x', n':)) = 2
{-2}	g`nodes(x')
{-3}	g`nodes(n')
{-4}	g`edges(x', n')
{1}	∀ (i: below(length((:x', n':)))): g`nodes(nth((:x', n':), i))

Repeatedly Skolemizing and flattening,

Case splitting on i!1 = 0,

we get 2 subgoals:

`non_root_path.1.2.1.1:`

{-1}	i' = 0
{-2}	i' < length((:x', n':))
{-3}	length$\lceil T \rceil$((:x', n':)) = 2
{-4}	g`nodes(x')
{-5}	g`nodes(n')
{-6}	g`edges(x', n')
{1}	g`nodes(nth((:x', n':), i'))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `non_root_path.1.2.1.1`.

non_root_path.1.2.1.2:

{-1}	$i' < \text{length}((:x', n':))$
{-2}	$\text{length}[T]((:x', n':)) = 2$
{-3}	$g'`\text{nodes}(x')$
{-4}	$g'`\text{nodes}(n')$
{-5}	$g'`\text{edges}(x', n')$
{1}	$i' = 0$
{2}	$g'`\text{nodes}(\text{nth}((:x', n':), i'))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of non_root_path.1.2.1.2.
non_root_path.1.2.2:

{-1}	$\text{length}[T]((:x', n':)) = 2$
{-2}	$g'`\text{nodes}(x')$
{-3}	$g'`\text{nodes}(n')$
{-4}	$g'`\text{edges}(x', n')$
{1}	$\forall\ (i\colon \text{below}(\text{length}((:x', n':)) - 1))\colon$ $g'`\text{edges}(\text{nth}((:x', n':), i), \text{nth}((:x', n':), 1 + i))$

Repeatedly Skolemizing and flattening,
Expanding the definition of nth,
Expanding the definition of nth,
which is trivially true.
This completes the proof of non_root_path.1.2.2.
non_root_path.2:

{-1}	$g'`\text{nodes}(x')$
{-2}	$g'`\text{nodes}(n')$
{-3}	$g'`\text{edges}(x', n')$
{1}	$\text{length}[T]((:x', n':)) = 2$
{2}	$\exists\ (p\colon (\text{path?}(g'\!)))\colon \text{length}(p) = 2 \wedge \text{path\_end}(p) = n'$

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of non_root_path.2.
Q.E.D.

C.97.22 Graph.non_root_path_extend_TCC1

Terse proof for non_root_path_extend_TCC1.
non_root_path_extend_TCC1:

{1}	$\forall\ (g\colon \text{graph\_type}, p\colon (\text{path?}(g)))\colon (g`\text{nodes})(\text{path\_start}[T](p))$

Repeatedly Skolemizing and flattening,
Rewriting using nodes_path_start, matching in *,
This completes the proof of non_root_path_extend_TCC1.
Q.E.D.

C.97.23 Graph.non_root_path_extend

Terse proof for non_root_path_extend.

`non_root_path_extend:`

{1} ∀ (g: graph_type, p: (path?(g))):
roots(g)(path_start(p)) ∨
(∃ (q: (path?(g))):
length(q) = length(p) + 1 ∧ path_end(q) = path_end(p))

Repeatedly Skolemizing and flattening,

Using lemma non_root_path,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: path_concat_path length_concat_path concatable?

Instantiating the top quantifier in 2 with the terms: concat_path(p'', p'),

we get 2 subgoals:

`non_root_path_extend.1:`

{-1} path_list?[T](p'')
{-2} path?(g')(p'')
{-3} length(p'') = 2
{-4} path_end(p'') = path_start(p')
{-5} path_list?[T](p')
{-6} path?(g')(p')

{1} roots(g')(path_start(p'))
{2} length(concat_path(p'', p')) = 1 + length(p') ∧
path_end(concat_path(p'', p')) = path_end(p')

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: path_end path_start nth_append_right nth_append_left concat_path length_append length_cdr path_list? length

Simplifying, rewriting, and recording with decision procedures,

Case splitting on length(p!1) = 1,

we get 2 subgoals:

`non_root_path_extend.1.1:`

{-1} length(p') = 1
{-2} path_list?[T](p'')
{-3} path?(g')(p'')
{-4} length(p'') = 2
{-5} nth(p'', length(p'') − 1) = car(p')
{-6} length(p') ≥ 1
{-7} path?(g')(p')

{1} roots(g')(car(p'))
{2} nth(append(p'', cdr(p')), length(p') + length(p'') − 2) =
nth(p', length(p') − 1)

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: nth

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `non_root_path_extend.1.1`.

`non_root_path_extend.1.2:`

{-1}	path_list?$[T]$(p'')
{-2}	path?(g')(p'')
{-3}	length(p'') = 2
{-4}	nth(p'', length(p'') − 1) = car(p')
{-5}	length(p') ≥ 1
{-6}	path?(g')(p')
{1}	length(p') = 1
{2}	roots(g')(car(p'))
{3}	nth(append(p'', cdr(p')), length(p') + length(p'') − 2) = nth(p', length(p') − 1)

Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: nth
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `non_root_path_extend.1.2`.
`non_root_path_extend.2:`

{-1}	path_list?$[T]$(p'')
{-2}	path?(g')(p'')
{-3}	length(p'') = 2
{-4}	path_end(p'') = path_start(p')
{-5}	path_list?$[T]$(p')
{-6}	path?(g')(p')
{1}	path?(g')(concat_path$[T]$(p'', p'))
{2}	roots(g')(path_start(p'))

Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `non_root_path_extend.2`.
Q.E.D.

C.97.24 Graph.tree_path_to_root

Terse proof for `tree_path_to_root`.
`tree_path_to_root:`

{1}	∀ (g: graph_type): finite_tree?(g) ⊃ (∀ (n: (g‘nodes)): ∃ (p: (path?(g))): root_path(g, n)(p))

Repeatedly Skolemizing and flattening,
Expanding the definition of finite_tree?,
Applying disjunctive simplification to flatten sequent,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of finite?,
Simplifying, rewriting, and recording with decision procedures,
Letting max_len name min(LAMBDA (i: nat): EXISTS (q: (path?(g!1))): path_end(q) = n!1 AND i = card(g!1‘nodes) - length(q)),
we get 2 subgoals:

`tree_path_to_root.1:`

{-1} $\min(\lambda\ (i\colon \text{nat})\colon \exists\ (q\colon (\text{path?}(g'))):\ \text{path\_end}(q) = n' \wedge i = \text{card}(g'\text{`nodes}) - \text{length}(q)) = \text{max\_len}$
{-2} $g'\text{`nodes}(n')$
{-3} $\text{is\_finite}(g'\text{`nodes})$
{-4} $\text{tree?}(g')$

{1} $\exists\ (p\colon (\text{path?}(g'))):\ \text{root\_path}(g', n')(p)$

Rewriting using min_def, matching in *,

Letting rp name choose(LAMBDA (p: (path?(g!1))): path_end(p) = n!1 AND length(p) = card(g!1`nodes) - max_len),

we get 2 subgoals:

`tree_path_to_root.1.1:`

{-1} $\text{choose}(\lambda\ (p\colon (\text{path?}(g'))):\ \text{path\_end}(p) = n' \wedge \text{length}(p) = \text{card}(g'\text{`nodes}) - \text{max\_len}) = \text{rp}$
{-2} $\text{minimum?}(\text{max\_len}, \lambda\ (i\colon \text{nat})\colon \exists\ (q\colon (\text{path?}(g'))):\ \text{path\_end}(q) = n' \wedge i = \text{card}(g'\text{`nodes}) - \text{length}(q))$
{-3} $g'\text{`nodes}(n')$
{-4} $\text{is\_finite}(g'\text{`nodes})$
{-5} $\text{tree?}(g')$

{1} $\exists\ (p\colon (\text{path?}(g'))):\ \text{root\_path}(g', n')(p)$

Instantiating the top quantifier in 1 with the terms: rp,

Expanding the definition of root_path,

Simplifying, rewriting, and recording with decision procedures,

Adding type constraints for rp,

Hiding formulas: -1, -3,

Using lemma non_root_path_extend,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Using lemma path_length_bound,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`tree_path_to_root.1.1.1:`

{-1}	$\forall\ (p\colon\ (\text{path?}(g'))):\ \text{length}(p)\ \leq\ \text{card}(g'\text{`nodes})$
{-2}	$\text{path\_list?}[T](q')$
{-3}	$\text{path?}(g')(q')$
{-4}	$\text{length}(q')\ =\ 1 + \text{length(rp)}$
{-5}	$\text{path\_end}(q')\ =\ \text{path\_end(rp)}$
{-6}	$\text{path?}(g')\text{(rp)}$
{-7}	$\text{length(rp)}\ =\ \text{card}(g'\text{`nodes}) - \text{max\_len}$
{-8}	$\text{choose}(\lambda\ (p\colon\ (\text{path?}(g'))):$ $\text{path\_end}(p)\ =\ n'\ \wedge\ \text{length}(p)\ =\ \text{card}(g'\text{`nodes}) - \text{max\_len})$ $=\ \text{rp}$
{-9}	minimum?(max_len, $\lambda\ (i\colon\ \text{nat}):$ $\exists\ (q\colon\ (\text{path?}(g'))):$ $\text{path\_end}(q)\ =\ n'\ \wedge\ i\ =\ \text{card}(g'\text{`nodes}) - \text{length}(q))$
{-10}	$g'\text{`nodes}(n')$
{-11}	$\text{is\_finite}(g'\text{`nodes})$
{-12}	$\text{tree?}(g')$
{1}	$\text{roots}(g')(\text{path\_start(rp)})$

Instantiating quantified variables,

Expanding the definition of minimum?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -8, -9,

Instantiating the top quantifier in -8 with the terms: $\text{card}(g'\text{`nodes}) - \text{length}(q')$,

we get 2 subgoals:

`tree_path_to_root.1.1.1.1:`

{-1}	$\text{length}(q')\ \leq\ \text{card}(g'\text{`nodes})$
{-2}	$\text{path\_list?}[T](q')$
{-3}	$\text{path?}(g')(q')$
{-4}	$\text{length}(q')\ =\ 1 + \text{length(rp)}$
{-5}	$\text{path\_end}(q')\ =\ \text{path\_end(rp)}$
{-6}	$\text{path?}(g')\text{(rp)}$
{-7}	$\text{length(rp)}\ =\ \text{card}(g'\text{`nodes}) - \text{max\_len}$
{-8}	$(\exists\ (q\colon\ (\text{path?}(g'))):$ $\text{path\_end}(q)\ =\ n'\ \wedge$ $\text{card}(g'\text{`nodes}) - \text{length}(q')\ =\ \text{card}(g'\text{`nodes}) - \text{length}(q))$ $\supset\ \text{max\_len}\ \leq\ \text{card}(g'\text{`nodes}) - \text{length}(q')$
{-9}	$g'\text{`nodes}(n')$
{-10}	$\text{is\_finite}(g'\text{`nodes})$
{-11}	$\text{tree?}(g')$
{1}	$\text{roots}(g')(\text{path\_start(rp)})$

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_path_to_root.1.1.1.1`.

`tree_path_to_root.1.1.1.2:`

{-1} length(q') $\leq$ card(g'`nodes)
{-2} path_list?$[T]$(q')
{-3} path?(g')(q')
{-4} length(q') = 1 + length(rp)
{-5} path_end(q') = path_end(rp)
{-6} path?(g')(rp)
{-7} length(rp) = card(g'`nodes) − max_len
{-8} g'`nodes(n')
{-9} is_finite(g'`nodes)
{-10} tree?(g')

{1} card$[T]$(g'`nodes) − length$[T]$(q') $\geq$ 0
{2} roots(g')(path_start(rp))

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path_to_root.1.1.1.2`.

`tree_path_to_root.1.1.2:`

{-1} path_list?$[T]$(q')
{-2} path?(g')(q')
{-3} length(q') = 1 + length(rp)
{-4} path_end(q') = path_end(rp)
{-5} path?(g')(rp)
{-6} length(rp) = card(g'`nodes) − max_len
{-7} choose(λ (p: (path?(g'))):
path_end(p) = n' $\wedge$ length(p) = card(g'`nodes) − max_len)
= rp
{-8} minimum?(max_len,
λ (i: nat):
$\exists$ (q: (path?(g'))):
path_end(q) = n' $\wedge$ i = card(g'`nodes) − length(q))
{-9} g'`nodes(n')
{-10} is_finite(g'`nodes)
{-11} tree?(g')

{1} cycle_free?(g')
{2} roots(g')(path_start(rp))

Expanding the definition of tree?,
which is trivially true.
This completes the proof of `tree_path_to_root.1.1.2`.

`tree_path_to_root.1.2:`

{-1} minimum?(max_len,
λ (i: nat):
$\exists$ (q: (path?(g'))):
path_end(q) = n' $\wedge$ i = card(g'`nodes) − length(q))
{-2} g'`nodes(n')
{-3} is_finite(g'`nodes)
{-4} tree?(g')

{1} nonempty?$[$(path?(g'))$]$
(λ (p: (path?(g'))):
path_end$[T]$(p) = n' $\wedge$
length$[T]$(p) = card$[T]$(g'`nodes) − max_len)
{2} $\exists$ (p: (path?(g'))): root_path(g', n')(p)

Expanding the definition of minimum?,
Applying disjunctive simplification to flatten sequent,
Hiding formulas: -2, 2,
Expanding the definition(s) of (nonempty? empty? member),
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path_to_root.1.2`.
`tree_path_to_root.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & g'\text{`nodes}(n') \\
\{\text{-}2\} & \text{is\_finite}(g'\text{`nodes}) \\
\{\text{-}3\} & \text{tree?}(g') \\
\hline
\{1\} & \text{nonempty?}[\text{nat}] \\
 & \quad (\lambda\ (i\colon \text{nat})\colon \\
 & \quad\quad \exists\ (q\colon (\text{path?}(g'))\colon \\
 & \quad\quad\quad \text{path\_end}[T](q) = n' \wedge i = \text{card}[T](g'\text{`nodes}) - \text{length}[T](q)) \\
\{2\} & \exists\ (p\colon (\text{path?}(g'))\colon \text{root\_path}(g', n')(p)
\end{array}
$$

Hiding formulas: 2,
Installing automatic rewrites from: nonempty? empty? member path_end nth length path_list? path?
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in -1 with the terms: $\text{card}(g'\text{`nodes}) - 1$,
we get 2 subgoals:
`tree_path_to_root.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & g'\text{`nodes}(n') \\
\{\text{-}2\} & \text{is\_finite}(g'\text{`nodes}) \\
\{\text{-}3\} & \text{tree?}(g') \\
\hline
\{1\} & \exists\ (q\colon (\text{path?}(g'))\colon \\
 & \quad \text{nth}(q, \text{length}(q) - 1) = n' \wedge \\
 & \quad\ \text{card}(g'\text{`nodes}) - 1 = \text{card}[T](g'\text{`nodes}) - \text{length}[T](q)
\end{array}
$$

Instantiating the top quantifier in 1 with the terms: $(:n':)$,
we get 2 subgoals:
`tree_path_to_root.2.1.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & g'\text{`nodes}(n') \\
\{\text{-}2\} & \text{is\_finite}(g'\text{`nodes}) \\
\{\text{-}3\} & \text{tree?}(g') \\
\hline
\{1\} & \text{nth}((:n':), \text{length}((:n':)) - 1) = n' \wedge \\
 & \quad \text{card}(g'\text{`nodes}) - 1 = \text{card}[T](g'\text{`nodes}) - \text{length}[T]((:n':))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_path_to_root.2.1.1`.
`tree_path_to_root.2.1.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & g'\text{`nodes}(n') \\
\{\text{-}2\} & \text{is\_finite}(g'\text{`nodes}) \\
\{\text{-}3\} & \text{tree?}(g') \\
\hline
\{1\} & (\forall\ (i\colon \text{below}(\text{length}((:n':))))\colon g'\text{`nodes}(\text{nth}((:n':), i))) \wedge \\
 & \quad (\forall\ (i\colon \text{below}(\text{length}((:n':)) - 1))\colon \\
 & \quad\quad g'\text{`edges}(\text{nth}((:n':), i), \text{nth}((::), i)))
\end{array}
$$

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `tree_path_to_root.2.1.2`.

`tree_path_to_root.2.2:`

{-1}	g'`'`nodes(n')
{-2}	is_finite(g'`'`nodes)
{-3}	tree?(g')
{1}	card$[T]$(g'`'`nodes) $-$ 1 $\geq$ 0

Using lemma card_singleton$[T]$,

Using lemma card_subset,

Simplifying, rewriting, and recording with decision procedures,

Using lemma singleton_subset$[T]$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_path_to_root.2.2`.

Q.E.D.

C.97.25 Graph.tree_unique_path_to_root

Terse proof for `tree_unique_path_to_root`.

`tree_unique_path_to_root:`

{1}	$\forall$ (g: graph_type, n: (g`'`nodes), p_1, p_2: (path?(g))): tree?(g) $\wedge$ root_path(g, n)(p_1) $\wedge$ root_path(g, n)(p_2) $\supset$ $p_1 = p_2$

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: path_list? length_tail car_tail cons_tail root_path

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on length(p1!1) < length(p2!1),

we get 2 subgoals:

`tree_unique_path_to_root.1:`

{-1}	length(p'_1) $<$ length(p'_2)
{-2}	g'`'`nodes(n')
{-3}	length(p'_1) $\geq$ 1
{-4}	path?(g')(p'_1)
{-5}	length(p'_2) $\geq$ 1
{-6}	path?(g')(p'_2)
{-7}	tree?(g')
{-8}	roots(g')(path_start(p'_1))
{-9}	path_end(p'_1) $=$ n'
{-10}	roots(g')(path_start(p'_2))
{-11}	path_end(p'_2) $=$ n'
{1}	$p'_1 = p'_2$

Using lemma path_tail,

we get 2 subgoals:

`tree_unique_path_to_root.1.1:`

{-1}	$\text{length}(p'_2) - \text{length}(p'_2) - \text{length}(p'_1) \geq 1 \supset$ $\text{path?}(g')(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-2}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-3}	$g' \text{`nodes}(n')$
{-4}	$\text{length}(p'_1) \geq 1$
{-5}	$\text{path?}(g')(p'_1)$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\text{path?}(g')(p'_2)$
{-8}	$\text{tree?}(g')$
{-9}	$\text{roots}(g')(\text{path\_start}(p'_1))$
{-10}	$\text{path\_end}(p'_1) = n'$
{-11}	$\text{roots}(g')(\text{path\_start}(p'_2))$
{-12}	$\text{path\_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,

Using lemma tree_path,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`tree_unique_path_to_root.1.1.1:`

{-1}	$\text{tree?}(g')$
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	$\text{path?}(g')(\text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1)))$
{-4}	$\text{length}(p'_1) < \text{length}(p'_2)$
{-5}	$g' \text{`nodes}(n')$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\text{path?}(g')(p'_1)$
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\text{path?}(g')(p'_2)$
{-10}	$\text{roots}(g')(\text{path\_start}(p'_1))$
{-11}	$\text{path\_end}(p'_1) = n'$
{-12}	$\text{roots}(g')(\text{path\_start}(p'_2))$
{-13}	$\text{path\_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Expanding the definition of roots,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -11 with the terms: $\text{nth}(p'_2, \text{length}(p'_2) - \text{length}(p'_1) - 1)$,

we get 2 subgoals:

`tree_unique_path_to_root.1.1.1.1:`

{-1}	tree?(g')
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	path?(g')(tail(p'_2, length(p'_2) − length(p'_1)))
{-4}	length(p'_1) < length(p'_2)
{-5}	g'`nodes(n')
{-6}	length(p'_1) ≥ 1
{-7}	path?(g')(p'_1)
{-8}	length(p'_2) ≥ 1
{-9}	∀ (i: below(length(p'_2))): g'`nodes(nth(p'_2, i))
{-10}	∀ (i: below(length(p'_2) − 1)): g'`edges(nth(p'_2, i), nth(p'_2, 1 + i))
{-11}	path_end(p'_1) = n'
{-12}	roots(g')(path_start(p'_2))
{-13}	path_end(p'_2) = n'
{1}	g'`edges(nth(p'_2, length(p'_2) − length(p'_1) − 1), path_start(p'_1))
{2}	$p'_1 = p'_2$

Instantiating the top quantifier in -10 with the terms: length(p'_2) − length(p'_1) − 1,

Expanding the definition of path_start,

Case splitting on car(p1!1) = car(tail(p2!1, length(p2!1) - length(p1!1))),

we get 2 subgoals:

`tree_unique_path_to_root.1.1.1.1.1:`

{-1}	car(p'_1) = car(tail(p'_2, length(p'_2) − length(p'_1)))
{-2}	tree?(g')
{-3}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-4}	path?(g')(tail(p'_2, length(p'_2) − length(p'_1)))
{-5}	length(p'_1) < length(p'_2)
{-6}	g'`nodes(n')
{-7}	length(p'_1) ≥ 1
{-8}	path?(g')(p'_1)
{-9}	length(p'_2) ≥ 1
{-10}	∀ (i: below(length(p'_2))): g'`nodes(nth(p'_2, i))
{-11}	g'`edges (nth(p'_2, length(p'_2) − length(p'_1) − 1), nth(p'_2, 1 + length(p'_2) − length(p'_1) − 1))
{-12}	path_end(p'_1) = n'
{-13}	roots(g')(path_start(p'_2))
{-14}	path_end(p'_2) = n'
{1}	g'`edges(nth(p'_2, length(p'_2) − 1 − length(p'_1)), car(p'_1))
{2}	$p'_1 = p'_2$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_unique_path_to_root.1.1.1.1.1`.

`tree_unique_path_to_root.1.1.1.1.2:`

{-1}	tree?(g')
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	path?(g')(tail(p'_2, length(p'_2) − length(p'_1)))
{-4}	length(p'_1) < length(p'_2)
{-5}	g'`nodes`(n')
{-6}	length(p'_1) ≥ 1
{-7}	path?(g')(p'_1)
{-8}	length(p'_2) ≥ 1
{-9}	∀ (i: below(length(p'_2))): g'`nodes`(nth(p'_2, i))
{-10}	g'`edges` (nth(p'_2, length(p'_2) − length(p'_1) − 1), nth(p'_2, 1 + length(p'_2) − length(p'_1) − 1))
{-11}	path_end(p'_1) = n'
{-12}	roots(g')(path_start(p'_2))
{-13}	path_end(p'_2) = n'
{1}	car(p'_1) = car(tail(p'_2, length(p'_2) − length(p'_1)))
{2}	g'`edges`(nth(p'_2, length(p'_2) − 1 − length(p'_1)), car(p'_1))
{3}	$p'_1 = p'_2$

Replacing using formula -2,

which is trivially true.

This completes the proof of `tree_unique_path_to_root.1.1.1.1.2`.

`tree_unique_path_to_root.1.1.1.2:`

{-1}	tree?(g')
{-2}	$p'_1 = \text{tail}(p'_2, \text{length}(p'_2) - \text{length}(p'_1))$
{-3}	path?(g')(tail(p'_2, length(p'_2) − length(p'_1)))
{-4}	length(p'_1) < length(p'_2)
{-5}	g'`nodes`(n')
{-6}	length(p'_1) ≥ 1
{-7}	path?(g')(p'_1)
{-8}	length(p'_2) ≥ 1
{-9}	∀ (i: below(length(p'_2))): g'`nodes`(nth(p'_2, i))
{-10}	∀ (i: below(length(p'_2) − 1)): g'`edges`(nth(p'_2, i), nth(p'_2, 1 + i))
{-11}	path_end(p'_1) = n'
{-12}	roots(g')(path_start(p'_2))
{-13}	path_end(p'_2) = n'
{1}	g'`nodes`(nth⌈T⌉(p'_2, length⌈T⌉(p'_2) − 1 − length⌈T⌉(p'_1)))
{2}	$p'_1 = p'_2$

Instantiating quantified variables,

This completes the proof of `tree_unique_path_to_root.1.1.1.2`.

`tree_unique_path_to_root.1.1.2:`

{-1}	tree?(g')
{-2}	path?(g')(tail(p'_2, length(p'_2) − length(p'_1)))
{-3}	length(p'_1) < length(p'_2)
{-4}	g'`nodes(n')
{-5}	length(p'_1) ≥ 1
{-6}	path?(g')(p'_1)
{-7}	length(p'_2) ≥ 1
{-8}	path?(g')(p'_2)
{-9}	roots(g')(path_start(p'_1))
{-10}	path_end(p'_1) = n'
{-11}	roots(g')(path_start(p'_2))
{-12}	path_end(p'_2) = n'
{1}	path_end(p'_1) = path_end(tail(p'_2, length(p'_2) − length(p'_1)))
{2}	p'_1 = p'_2

Installing automatic rewrites from: path_end nth_tail
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_unique_path_to_root.1.1.2`.
`tree_unique_path_to_root.1.2:`

{-1}	length(p'_1) < length(p'_2)
{-2}	g'`nodes(n')
{-3}	length(p'_1) ≥ 1
{-4}	path?(g')(p'_1)
{-5}	length(p'_2) ≥ 1
{-6}	path?(g')(p'_2)
{-7}	tree?(g')
{-8}	roots(g')(path_start(p'_1))
{-9}	path_end(p'_1) = n'
{-10}	roots(g')(path_start(p'_2))
{-11}	path_end(p'_2) = n'
{1}	length⌈T⌉(p'_2) − length⌈T⌉(p'_1) ≥ 0
{2}	p'_1 = p'_2

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_unique_path_to_root.1.2`.
`tree_unique_path_to_root.2:`

{-1}	g'`nodes(n')
{-2}	length(p'_1) ≥ 1
{-3}	path?(g')(p'_1)
{-4}	length(p'_2) ≥ 1
{-5}	path?(g')(p'_2)
{-6}	tree?(g')
{-7}	roots(g')(path_start(p'_1))
{-8}	path_end(p'_1) = n'
{-9}	roots(g')(path_start(p'_2))
{-10}	path_end(p'_2) = n'
{1}	length(p'_1) < length(p'_2)
{2}	p'_1 = p'_2

Case splitting on length(p1!1) > length(p2!1),
we get 2 subgoals:

`tree_unique_path_to_root.2.1:`

{-1}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-2}	$g'\text{`nodes}(n')$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{tree?}(g')$
{-8}	$\text{roots}(g')(\text{path\_start}(p'_1))$
{-9}	$\text{path\_end}(p'_1) = n'$
{-10}	$\text{roots}(g')(\text{path\_start}(p'_2))$
{-11}	$\text{path\_end}(p'_2) = n'$
{1}	$\text{length}(p'_1) < \text{length}(p'_2)$
{2}	$p'_1 = p'_2$

Hiding formulas: 1,

Using lemma path_tail,

we get 2 subgoals:

`tree_unique_path_to_root.2.1.1:`

{-1}	$\text{length}(p'_1) - \text{length}(p'_1) - \text{length}(p'_2) \geq 1 \supset$ $\text{path?}(g')(\text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2)))$
{-2}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-3}	$g'\text{`nodes}(n')$
{-4}	$\text{length}(p'_1) \geq 1$
{-5}	$\text{path?}(g')(p'_1)$
{-6}	$\text{length}(p'_2) \geq 1$
{-7}	$\text{path?}(g')(p'_2)$
{-8}	$\text{tree?}(g')$
{-9}	$\text{roots}(g')(\text{path\_start}(p'_1))$
{-10}	$\text{path\_end}(p'_1) = n'$
{-11}	$\text{roots}(g')(\text{path\_start}(p'_2))$
{-12}	$\text{path\_end}(p'_2) = n'$
{1}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,

Using lemma tree_path,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`tree_unique_path_to_root.2.1.1.1:`

{-1}	tree?(g')
{-2}	$p'_2 = \text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2))$
{-3}	path?(g')(tail(p'_1, length(p'_1) − length(p'_2)))
{-4}	length(p'_1) > length(p'_2)
{-5}	g'`nodes(n')
{-6}	length(p'_1) ≥ 1
{-7}	path?(g')(p'_1)
{-8}	length(p'_2) ≥ 1
{-9}	path?(g')(p'_2)
{-10}	roots(g')(path_start(p'_1))
{-11}	path_end(p'_1) = n'
{-12}	roots(g')(path_start(p'_2))
{-13}	path_end(p'_2) = n'
{1}	$p'_1 = p'_2$

Expanding the definition of roots,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -13 with the terms: nth(p'_1, length(p'_1) − length(p'_2) − 1),

we get 2 subgoals:

`tree_unique_path_to_root.2.1.1.1.1:`

{-1}	tree?(g')
{-2}	$p'_2 = \text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2))$
{-3}	path?(g')(tail(p'_1, length(p'_1) − length(p'_2)))
{-4}	length(p'_1) > length(p'_2)
{-5}	g'`nodes(n')
{-6}	length(p'_1) ≥ 1
{-7}	∀ (i: below(length(p'_1))): g'`nodes(nth(p'_1, i))
{-8}	∀ (i: below(length(p'_1) − 1)): g'`edges(nth(p'_1, i), nth(p'_1, 1 + i))
{-9}	length(p'_2) ≥ 1
{-10}	path?(g')(p'_2)
{-11}	roots(g')(path_start(p'_1))
{-12}	path_end(p'_1) = n'
{-13}	path_end(p'_2) = n'
{1}	g'`edges(nth(p'_1, length(p'_1) − length(p'_2) − 1), path_start(p'_2))
{2}	$p'_1 = p'_2$

Instantiating the top quantifier in -8 with the terms: length(p'_1) − length(p'_2) − 1,

Expanding the definition of path_start,

Case splitting on car(p2!1) = car(tail(p1!1, length(p1!1) - length(p2!1))),

we get 2 subgoals:

`tree_unique_path_to_root.2.1.1.1.1.1:`

{-1}	$\mathrm{car}(p'_2) = \mathrm{car}(\mathrm{tail}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2)))$
{-2}	$\mathrm{tree?}(g')$
{-3}	$p'_2 = \mathrm{tail}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2))$
{-4}	$\mathrm{path?}(g')(\mathrm{tail}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2)))$
{-5}	$\mathrm{length}(p'_1) > \mathrm{length}(p'_2)$
{-6}	$g'\text{`}\mathrm{nodes}(n')$
{-7}	$\mathrm{length}(p'_1) \geq 1$
{-8}	$\forall\ (i\colon \mathrm{below}(\mathrm{length}(p'_1)))\colon\ g'\text{`}\mathrm{nodes}(\mathrm{nth}(p'_1, i))$
{-9}	$g'\text{`}\mathrm{edges}$ $(\mathrm{nth}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2) - 1),$ $\mathrm{nth}(p'_1, 1 + \mathrm{length}(p'_1) - \mathrm{length}(p'_2) - 1))$
{-10}	$\mathrm{length}(p'_2) \geq 1$
{-11}	$\mathrm{path?}(g')(p'_2)$
{-12}	$\mathrm{roots}(g')(\mathrm{path\_start}(p'_1))$
{-13}	$\mathrm{path\_end}(p'_1) = n'$
{-14}	$\mathrm{path\_end}(p'_2) = n'$
{1}	$g'\text{`}\mathrm{edges}(\mathrm{nth}(p'_1, \mathrm{length}(p'_1) - 1 - \mathrm{length}(p'_2)), \mathrm{car}(p'_2))$
{2}	$p'_1 = p'_2$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_unique_path_to_root.2.1.1.1.1.1`.

`tree_unique_path_to_root.2.1.1.1.1.2:`

{-1}	$\mathrm{tree?}(g')$
{-2}	$p'_2 = \mathrm{tail}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2))$
{-3}	$\mathrm{path?}(g')(\mathrm{tail}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2)))$
{-4}	$\mathrm{length}(p'_1) > \mathrm{length}(p'_2)$
{-5}	$g'\text{`}\mathrm{nodes}(n')$
{-6}	$\mathrm{length}(p'_1) \geq 1$
{-7}	$\forall\ (i\colon \mathrm{below}(\mathrm{length}(p'_1)))\colon\ g'\text{`}\mathrm{nodes}(\mathrm{nth}(p'_1, i))$
{-8}	$g'\text{`}\mathrm{edges}$ $(\mathrm{nth}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2) - 1),$ $\mathrm{nth}(p'_1, 1 + \mathrm{length}(p'_1) - \mathrm{length}(p'_2) - 1))$
{-9}	$\mathrm{length}(p'_2) \geq 1$
{-10}	$\mathrm{path?}(g')(p'_2)$
{-11}	$\mathrm{roots}(g')(\mathrm{path\_start}(p'_1))$
{-12}	$\mathrm{path\_end}(p'_1) = n'$
{-13}	$\mathrm{path\_end}(p'_2) = n'$
{1}	$\mathrm{car}(p'_2) = \mathrm{car}(\mathrm{tail}(p'_1, \mathrm{length}(p'_1) - \mathrm{length}(p'_2)))$
{2}	$g'\text{`}\mathrm{edges}(\mathrm{nth}(p'_1, \mathrm{length}(p'_1) - 1 - \mathrm{length}(p'_2)), \mathrm{car}(p'_2))$
{3}	$p'_1 = p'_2$

Replacing using formula -2,

which is trivially true.

This completes the proof of `tree_unique_path_to_root.2.1.1.1.1.2`.

`tree_unique_path_to_root.2.1.1.1.2:`

{-1}	tree?(g')
{-2}	$p'_2 = \text{tail}(p'_1, \text{length}(p'_1) - \text{length}(p'_2))$
{-3}	path?(g')(tail(p'_1, length(p'_1) − length(p'_2)))
{-4}	length(p'_1) > length(p'_2)
{-5}	g'`nodes(n')
{-6}	length(p'_1) ≥ 1
{-7}	∀ (i: below(length(p'_1))): g'`nodes(nth(p'_1, i))
{-8}	∀ (i: below(length(p'_1) − 1)): g'`edges(nth(p'_1, i), nth(p'_1, 1 + i))
{-9}	length(p'_2) ≥ 1
{-10}	path?(g')(p'_2)
{-11}	roots(g')(path_start(p'_1))
{-12}	path_end(p'_1) = n'
{-13}	path_end(p'_2) = n'
{1}	g'`nodes(nth⌊T⌋(p'_1, length⌊T⌋(p'_1) − 1 − length⌊T⌋(p'_2)))
{2}	$p'_1 = p'_2$

Instantiating quantified variables,

This completes the proof of `tree_unique_path_to_root.2.1.1.1.2`.

`tree_unique_path_to_root.2.1.1.2:`

{-1}	tree?(g')
{-2}	path?(g')(tail(p'_1, length(p'_1) − length(p'_2)))
{-3}	length(p'_1) > length(p'_2)
{-4}	g'`nodes(n')
{-5}	length(p'_1) ≥ 1
{-6}	path?(g')(p'_1)
{-7}	length(p'_2) ≥ 1
{-8}	path?(g')(p'_2)
{-9}	roots(g')(path_start(p'_1))
{-10}	path_end(p'_1) = n'
{-11}	roots(g')(path_start(p'_2))
{-12}	path_end(p'_2) = n'
{1}	path_end(p'_2) = path_end(tail(p'_1, length(p'_1) − length(p'_2)))
{2}	$p'_1 = p'_2$

Installing automatic rewrites from: path_end nth_tail

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `tree_unique_path_to_root.2.1.1.2`.

`tree_unique_path_to_root.2.1.2:`

{-1}	$\text{length}(p'_1) > \text{length}(p'_2)$
{-2}	$g'\text{`nodes}(n')$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{tree?}(g')$
{-8}	$\text{roots}(g')(\text{path\_start}(p'_1))$
{-9}	$\text{path\_end}(p'_1) = n'$
{-10}	$\text{roots}(g')(\text{path\_start}(p'_2))$
{-11}	$\text{path\_end}(p'_2) = n'$
{1}	$\text{length}[T](p'_1) - \text{length}[T](p'_2) \geq 0$
{2}	$p'_1 = p'_2$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `tree_unique_path_to_root.2.1.2`.
`tree_unique_path_to_root.2.2:`

{-1}	$g'\text{`nodes}(n')$
{-2}	$\text{length}(p'_1) \geq 1$
{-3}	$\text{path?}(g')(p'_1)$
{-4}	$\text{length}(p'_2) \geq 1$
{-5}	$\text{path?}(g')(p'_2)$
{-6}	$\text{tree?}(g')$
{-7}	$\text{roots}(g')(\text{path\_start}(p'_1))$
{-8}	$\text{path\_end}(p'_1) = n'$
{-9}	$\text{roots}(g')(\text{path\_start}(p'_2))$
{-10}	$\text{path\_end}(p'_2) = n'$
{1}	$\text{length}(p'_1) > \text{length}(p'_2)$
{2}	$\text{length}(p'_1) < \text{length}(p'_2)$
{3}	$p'_1 = p'_2$

Rewriting using tree_path, matching in *,
This completes the proof of `tree_unique_path_to_root.2.2`.
Q.E.D.

C.97.26 Graph.root_path_singleton

Terse proof for `root_path_singleton`.
`root_path_singleton:`

{1}	$\forall$ (g: graph_type, n: (g`nodes)): finite_tree?(g) $\supset$ singleton?(root_path(g, n))

Repeatedly Skolemizing and flattening,
Expanding the definition of singleton?,
Using lemma tree_path_to_root,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly Skolemizing and flattening,

Using lemma tree_unique_path_to_root,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of finite_tree?,
which is trivially true.
This completes the proof of `root_path_singleton`.
Q.E.D.

C.97.27 Graph.path_to_root_TCC1

Terse proof for `path_to_root_TCC1`.

`path_to_root_TCC1`:

```
{1}  ∀ (g: (finite_tree?), n: (g`nodes)):
       singleton?[((path?(g)))](root_path(g, n))
```

Repeatedly Skolemizing and flattening,
Rewriting using root_path_singleton, matching in *,
This completes the proof of `path_to_root_TCC1`.
Q.E.D.

C.97.28 Graph.path_factor_TCC1

Terse proof for `path_factor_TCC1`.

`path_factor_TCC1`:

```
{1}  ∀ (g: graph_type, p1, p2: (path?(g))):
       (path_start(p1) = path_start(p2) ∧
          (∃ (n: nat):
               n < length(p1) ∧ n < length(p2) ∧ ¬ nth(p1, n) = nth(p2, n)))
        ⊃
        (∀ (q, p1t, p2t: (path?(g))):
             p2 = concat_path(q, p2t) ∧
              p1 = concat_path(q, p1t) ∧
               length(p2t) ≥ 2 ∧
                 length(p1t) ≥ 2 ∧ concatable?(q, p2t) ∧ concatable?(q, p1t)
               ⊃ 0 < length[T](p1t))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path_factor_TCC1`.
Q.E.D.

C.97.29 Graph.path_factor_TCC2

Terse proof for `path_factor_TCC2`.

path_factor_TCC2:

{1} ∀ (g: graph_type, p_1, p_2: (path?(g))):
(path_start(p_1) = path_start(p_2) ∧
(∃ (n: nat):
n < length(p_1) ∧ n < length(p_2) ∧ ¬ nth(p_1, n) = nth(p_2, n)))
⊃
(∀ (q, p1t, p2t: (path?(g))):
p_2 = concat_path(q, p2t) ∧
p_1 = concat_path(q, p1t) ∧
length(p2t) ≥ 2 ∧
length(p1t) ≥ 2 ∧ concatable?(q, p2t) ∧ concatable?(q, p1t)
⊃ 0 < length[T](p2t))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_factor_TCC2.

Q.E.D.

C.97.30 Graph.path_factor_TCC3

Terse proof for path_factor_TCC3.

path_factor_TCC3:

{1} ∀ (g: graph_type, p_1, p_2: (path?(g))):
(path_start(p_1) = path_start(p_2) ∧
(∃ (n: nat):
n < length(p_1) ∧ n < length(p_2) ∧ ¬ nth(p_1, n) = nth(p_2, n)))
⊃
(∀ (q, p1t, p2t: (path?(g))):
nth(p1t, 0) = nth(p2t, 0) ∧
p_2 = concat_path(q, p2t) ∧
p_1 = concat_path(q, p1t) ∧
length(p2t) ≥ 2 ∧
length(p1t) ≥ 2 ∧ concatable?(q, p2t) ∧ concatable?(q, p1t)
⊃ 1 < length[T](p1t))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of path_factor_TCC3.

Q.E.D.

C.97.31 Graph.path_factor_TCC4

Terse proof for path_factor_TCC4.

`path_factor_TCC4:`

{1} ∀ (g: graph_type, p_1, p_2: (path?(g))):
(path_start(p_1) = path_start(p_2) ∧
(∃ (n: nat):
n < length(p_1) ∧ n < length(p_2) ∧ ¬ nth(p_1, n) = nth(p_2, n)))
⊃
(∀ (q, p1t, p2t: (path?(g))):
nth(p1t, 0) = nth(p2t, 0) ∧
p_2 = concat_path(q, p2t) ∧
p_1 = concat_path(q, p1t) ∧
length(p2t) ≥ 2 ∧
length(p1t) ≥ 2 ∧ concatable?(q, p2t) ∧ concatable?(q, p1t)
⊃ 1 < length[T](p2t))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `path_factor_TCC4`.

Q.E.D.

C.97.32 Graph.path_factor

Terse proof for `path_factor`.

`path_factor:`

{1} ∀ (g: graph_type, p_1, p_2: (path?(g))):
(path_start(p_1) = path_start(p_2) ∧
(∃ (n: nat):
n < length(p_1) ∧ n < length(p_2) ∧ ¬ nth(p_1, n) = nth(p_2, n)))
⊃
(∃ (q, p1t, p2t: (path?(g))):
concatable?(q, p1t) ∧
concatable?(q, p2t) ∧
length(p1t) ≥ 2 ∧
length(p2t) ≥ 2 ∧
p_1 = concat_path(q, p1t) ∧
p_2 = concat_path(q, p2t) ∧
nth(p1t, 0) = nth(p2t, 0) ∧ ¬ nth(p1t, 1) = nth(p2t, 1))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: concatable? concat_path head_tail path_list? length_head length_tail path_end path_start nth_tail car_tail cdr_tail path_tail path_head nth_head

Simplifying, rewriting, and recording with decision procedures,

Letting nd name min(LAMBDA (i: nat): i < length(p1!1) AND i < length(p2!1) AND NOT nth(p1!1, i) = nth(p2!1, i)),

we get 2 subgoals:

`path_factor.1:`

{-1} $\min(\lambda\ (i\colon \text{nat}):\ i < \text{length}(p_1') \wedge i < \text{length}(p_2') \wedge \neg\ \text{nth}(p_1', i) = \text{nth}(p_2', i)) = \text{nd}$
{-2} $n' \geq 0$
{-3} $\text{length}(p_1') \geq 1$
{-4} $\text{path?}(g')(p_1')$
{-5} $\text{length}(p_2') \geq 1$
{-6} $\text{path?}(g')(p_2')$
{-7} $\text{car}(p_1') = \text{car}(p_2')$
{-8} $n' < \text{length}(p_1')$
{-9} $n' < \text{length}(p_2')$

{1} $\text{nth}(p_1', n') = \text{nth}(p_2', n')$
{2} $\exists\ (q, \text{p1t}, \text{p2t}\colon (\text{path?}(g'))):\ \text{nth}(q, \text{length}(q) - 1) = \text{car}(\text{p1t}) \wedge \text{nth}(q, \text{length}(q) - 1) = \text{car}(\text{p2t}) \wedge \text{length}(\text{p1t}) \geq 2 \wedge \text{length}(\text{p2t}) \geq 2 \wedge p_1' = \text{append}(q, \text{cdr}(\text{p1t})) \wedge p_2' = \text{append}(q, \text{cdr}(\text{p2t})) \wedge \text{nth}(\text{p1t}, 0) = \text{nth}(\text{p2t}, 0) \wedge \neg\ \text{nth}(\text{p1t}, 1) = \text{nth}(\text{p2t}, 1)$

Rewriting using min_def, matching in *,
Case splitting on nd = 0,
we get 2 subgoals:
`path_factor.1.1:`

{-1} $\text{nd} = 0$
{-2} $\text{minimum?}(\text{nd}, \lambda\ (i\colon \text{nat}):\ i < \text{length}(p_1') \wedge i < \text{length}(p_2') \wedge \neg\ \text{nth}(p_1', i) = \text{nth}(p_2', i))$
{-3} $n' \geq 0$
{-4} $\text{length}(p_1') \geq 1$
{-5} $\text{path?}(g')(p_1')$
{-6} $\text{length}(p_2') \geq 1$
{-7} $\text{path?}(g')(p_2')$
{-8} $\text{car}(p_1') = \text{car}(p_2')$
{-9} $n' < \text{length}(p_1')$
{-10} $n' < \text{length}(p_2')$

{1} $\text{nth}(p_1', n') = \text{nth}(p_2', n')$
{2} $\exists\ (q, \text{p1t}, \text{p2t}\colon (\text{path?}(g'))):\ \text{nth}(q, \text{length}(q) - 1) = \text{car}(\text{p1t}) \wedge \text{nth}(q, \text{length}(q) - 1) = \text{car}(\text{p2t}) \wedge \text{length}(\text{p1t}) \geq 2 \wedge \text{length}(\text{p2t}) \geq 2 \wedge p_1' = \text{append}(q, \text{cdr}(\text{p1t})) \wedge p_2' = \text{append}(q, \text{cdr}(\text{p2t})) \wedge \text{nth}(\text{p1t}, 0) = \text{nth}(\text{p2t}, 0) \wedge \neg\ \text{nth}(\text{p1t}, 1) = \text{nth}(\text{p2t}, 1)$

Hiding formulas: -2, 2,
Adding type constraints for nd,
Installing automatic rewrites from: nth
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.1`.

`path_factor.1.2:`

{-1} minimum?(nd,
  λ (i: nat):
    $i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg\ \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$)
{-2} $n' \geq 0$
{-3} $\text{length}(p'_1) \geq 1$
{-4} $\text{path?}(g')(p'_1)$
{-5} $\text{length}(p'_2) \geq 1$
{-6} $\text{path?}(g')(p'_2)$
{-7} $\text{car}(p'_1) = \text{car}(p'_2)$
{-8} $n' < \text{length}(p'_1)$
{-9} $n' < \text{length}(p'_2)$

{1} nd = 0
{2} $\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$
{3} ∃ (q, p1t, p2t: (path?(g'))):
  nth(q, length(q) − 1) = car(p1t) ∧
   nth(q, length(q) − 1) = car(p2t) ∧
    length(p1t) ≥ 2 ∧
     length(p2t) ≥ 2 ∧
      $p'_1 = \text{append}(q, \text{cdr}(\text{p1t}))$ ∧
       $p'_2 = \text{append}(q, \text{cdr}(\text{p2t}))$ ∧
        nth(p1t, 0) = nth(p2t, 0) ∧ ¬ nth(p1t, 1) = nth(p2t, 1)

Instantiating the top quantifier in 3 with the terms: $\text{head}(p'_1, \text{nd})$, $\text{tail}(p'_1, \text{nd} - 1)$, $\text{tail}(p'_2, \text{nd} - 1)$, we get 6 subgoals:

`path_factor.1.2.1:`

{-1} minimum?(nd,
  λ (i: nat):
    $i < \text{length}(p'_1) \wedge i < \text{length}(p'_2) \wedge \neg\ \text{nth}(p'_1, i) = \text{nth}(p'_2, i)$)
{-2} $n' \geq 0$
{-3} $\text{length}(p'_1) \geq 1$
{-4} $\text{path?}(g')(p'_1)$
{-5} $\text{length}(p'_2) \geq 1$
{-6} $\text{path?}(g')(p'_2)$
{-7} $\text{car}(p'_1) = \text{car}(p'_2)$
{-8} $n' < \text{length}(p'_1)$
{-9} $n' < \text{length}(p'_2)$

{1} nd = 0
{2} $\text{nth}(p'_1, n') = \text{nth}(p'_2, n')$
{3} $\text{nth}(\text{head}(p'_1, \text{nd}), \text{length}(\text{head}(p'_1, \text{nd})) - 1) =$
 $\text{car}(\text{tail}(p'_1, \text{nd} - 1))$
 ∧
 $\text{nth}(\text{head}(p'_1, \text{nd}), \text{length}(\text{head}(p'_1, \text{nd})) - 1) =$
  $\text{car}(\text{tail}(p'_2, \text{nd} - 1))$
  ∧
  $\text{length}(\text{tail}(p'_1, \text{nd} - 1)) \geq 2 \wedge$
   $\text{length}(\text{tail}(p'_2, \text{nd} - 1)) \geq 2 \wedge$
    $p'_1 = \text{append}(\text{head}(p'_1, \text{nd}), \text{cdr}(\text{tail}(p'_1, \text{nd} - 1))) \wedge$
     $p'_2 = \text{append}(\text{head}(p'_1, \text{nd}), \text{cdr}(\text{tail}(p'_2, \text{nd} - 1))) \wedge$
      $\text{nth}(\text{tail}(p'_1, \text{nd} - 1), 0) = \text{nth}(\text{tail}(p'_2, \text{nd} - 1), 0) \wedge$
       $\neg\ \text{nth}(\text{tail}(p'_1, \text{nd} - 1), 1) = \text{nth}(\text{tail}(p'_2, \text{nd} - 1), 1)$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of minimum?,

Applying propositional simplification,

we get 3 subgoals:

`path_factor.1.2.1.1:`

{-1}	$\forall$ (x: nat): $x < \text{length}(p_1') \wedge x < \text{length}(p_2') \wedge \neg\ \text{nth}(p_1', x) = \text{nth}(p_2', x) \supset \text{nd} \leq x$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p_1') \geq 1$
{-4}	$\text{path?}(g')(p_1')$
{-5}	$\text{length}(p_2') \geq 1$
{-6}	$\text{path?}(g')(p_2')$
{-7}	$\text{car}(p_1') = \text{car}(p_2')$
{-8}	$n' < \text{length}(p_1')$
{-9}	$n' < \text{length}(p_2')$
{1}	$\text{nth}(p_1', \text{nd} - 1) = \text{nth}(p_2', \text{nd} - 1)$
{2}	$\text{nd} = 0$
{3}	$\text{nth}(p_1', n') = \text{nth}(p_2', n')$

Instantiating the top quantifier in -1 with the terms: $\text{nd} - 1$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.1.1`.

`path_factor.1.2.1.2:`

{-1}	$\forall$ (x: nat): $x < \text{length}(p_1') \wedge x < \text{length}(p_2') \wedge \neg\ \text{nth}(p_1', x) = \text{nth}(p_2', x) \supset \text{nd} \leq x$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p_1') \geq 1$
{-4}	$\text{path?}(g')(p_1')$
{-5}	$\text{length}(p_2') \geq 1$
{-6}	$\text{path?}(g')(p_2')$
{-7}	$\text{car}(p_1') = \text{car}(p_2')$
{-8}	$n' < \text{length}(p_1')$
{-9}	$n' < \text{length}(p_2')$
{1}	$p_2' = \text{append}(\text{head}(p_1', \text{nd}), \text{tail}(p_2', \text{nd}))$
{2}	$\text{nd} = 0$
{3}	$\text{nth}(p_1', n') = \text{nth}(p_2', n')$

Using lemma list_extensionality,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`path_factor.1.2.1.2.1:`

{-1}	$\text{head}(p'_1,\ \text{nd}) = \text{head}(p'_2,\ \text{nd})$
{-2}	$\text{nd} = \text{nd}$
{-3}	$\forall\ (i\colon\ \text{below}(\text{length}[T](\text{head}(p'_1,\ \text{nd})))):$ $\text{nth}(\text{head}(p'_1,\ \text{nd}),\ i) = \text{nth}(\text{head}(p'_2,\ \text{nd}),\ i)$
{-4}	$\forall\ (x\colon\ \text{nat}):$ $x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg\ \text{nth}(p'_1,\ x) = \text{nth}(p'_2,\ x) \supset \text{nd} \leq x$
{-5}	$n' \geq 0$
{-6}	$\text{length}(p'_1) \geq 1$
{-7}	$\text{path?}(g')(p'_1)$
{-8}	$\text{length}(p'_2) \geq 1$
{-9}	$\text{path?}(g')(p'_2)$
{-10}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-11}	$n' < \text{length}(p'_1)$
{-12}	$n' < \text{length}(p'_2)$
{1}	$p'_2 = \text{append}(\text{head}(p'_1,\ \text{nd}),\ \text{tail}(p'_2,\ \text{nd}))$
{2}	$\text{nd} = 0$
{3}	$\text{nth}(p'_1,\ n') = \text{nth}(p'_2,\ n')$

Replacing using formula -1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.1.2.1`.

`path_factor.1.2.1.2.2:`

{-1}	$\forall\ (x\colon\ \text{nat}):$ $x < \text{length}(p'_1) \wedge x < \text{length}(p'_2) \wedge \neg\ \text{nth}(p'_1,\ x) = \text{nth}(p'_2,\ x) \supset \text{nd} \leq x$
{-2}	$n' \geq 0$
{-3}	$\text{length}(p'_1) \geq 1$
{-4}	$\text{path?}(g')(p'_1)$
{-5}	$\text{length}(p'_2) \geq 1$
{-6}	$\text{path?}(g')(p'_2)$
{-7}	$\text{car}(p'_1) = \text{car}(p'_2)$
{-8}	$n' < \text{length}(p'_1)$
{-9}	$n' < \text{length}(p'_2)$
{1}	$\text{head}(p'_1,\ \text{nd}) = \text{head}(p'_2,\ \text{nd})$
{2}	$\forall\ (i\colon\ \text{below}(\text{length}[T](\text{head}(p'_1,\ \text{nd})))):$ $\text{nth}(\text{head}(p'_1,\ \text{nd}),\ i) = \text{nth}(\text{head}(p'_2,\ \text{nd}),\ i)$
{3}	$p'_2 = \text{append}(\text{head}(p'_1,\ \text{nd}),\ \text{tail}(p'_2,\ \text{nd}))$
{4}	$\text{nd} = 0$
{5}	$\text{nth}(p'_1,\ n') = \text{nth}(p'_2,\ n')$

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -2 with the terms: i',

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.1.2.2`.

`path_factor.1.2.1.3:`

```
{-1}  ∀ (x: nat):
        x < length(p'1) ∧ x < length(p'2) ∧ ¬ nth(p'1, x) = nth(p'2, x) ⊃ nd ≤ x
{-2}  n' ≥ 0
{-3}  length(p'1) ≥ 1
{-4}  path?(g')(p'1)
{-5}  length(p'2) ≥ 1
{-6}  path?(g')(p'2)
{-7}  car(p'1) = car(p'2)
{-8}  n' < length(p'1)
{-9}  n' < length(p'2)
------------------------------------------------------------
{1}   nth(p'1, nd − 1) = nth(p'2, nd − 1)
{2}   nd = 0
{3}   nth(p'1, n') = nth(p'2, n')
```

Instantiating the top quantifier in -1 with the terms: nd − 1,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.1.2.1.3`.
`path_factor.1.2.2:`

```
{-1}  minimum?(nd,
               λ (i: nat):
                 i < length(p'1) ∧ i < length(p'2) ∧ ¬ nth(p'1, i) = nth(p'2, i))
{-2}  n' ≥ 0
{-3}  length(p'1) ≥ 1
{-4}  path?(g')(p'1)
{-5}  length(p'2) ≥ 1
{-6}  path?(g')(p'2)
{-7}  car(p'1) = car(p'2)
{-8}  n' < length(p'1)
{-9}  n' < length(p'2)
------------------------------------------------------------
{1}   length(tail[T](p'2, nd − 1)) ≥ 1 ∧
        path?(g')(tail[T](p'2, nd − 1))
{2}   nd = 0
{3}   nth(p'1, n') = nth(p'2, n')
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.1.2.2`.
`path_factor.1.2.3:`

```
{-1}  minimum?(nd,
               λ (i: nat):
                 i < length(p'1) ∧ i < length(p'2) ∧ ¬ nth(p'1, i) = nth(p'2, i))
{-2}  n' ≥ 0
{-3}  length(p'1) ≥ 1
{-4}  path?(g')(p'1)
{-5}  length(p'2) ≥ 1
{-6}  path?(g')(p'2)
{-7}  car(p'1) = car(p'2)
{-8}  n' < length(p'1)
{-9}  n' < length(p'2)
------------------------------------------------------------
{1}   nd − 1 ≥ 0
{2}   nd = 0
{3}   nth(p'1, n') = nth(p'2, n')
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.3`.

`path_factor.1.2.4:`

```
{-1}  minimum?(nd,
                λ (i: nat):
                  i < length(p'1) ∧ i < length(p'2) ∧ ¬ nth(p'1, i) = nth(p'2, i))
{-2}  n' ≥ 0
{-3}  length(p'1) ≥ 1
{-4}  path?(g')(p'1)
{-5}  length(p'2) ≥ 1
{-6}  path?(g')(p'2)
{-7}  car(p'1) = car(p'2)
{-8}  n' < length(p'1)
{-9}  n' < length(p'2)
------------------------------------------------------------------------
{1}   length(tail[T](p'1, nd − 1)) ≥ 1 ∧
       path?(g')(tail[T](p'1, nd − 1))
{2}   nd = 0
{3}   nth(p'1, n') = nth(p'2, n')
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.4`.

`path_factor.1.2.5:`

```
{-1}  minimum?(nd,
                λ (i: nat):
                  i < length(p'1) ∧ i < length(p'2) ∧ ¬ nth(p'1, i) = nth(p'2, i))
{-2}  n' ≥ 0
{-3}  length(p'1) ≥ 1
{-4}  path?(g')(p'1)
{-5}  length(p'2) ≥ 1
{-6}  path?(g')(p'2)
{-7}  car(p'1) = car(p'2)
{-8}  n' < length(p'1)
{-9}  n' < length(p'2)
------------------------------------------------------------------------
{1}   nd − 1 ≥ 0
{2}   nd = 0
{3}   nth(p'1, n') = nth(p'2, n')
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `path_factor.1.2.5`.

`path_factor.1.2.6`:

```
{-1}  minimum?(nd,
                λ (i: nat):
                  i < length(p'_1) ∧ i < length(p'_2) ∧ ¬ nth(p'_1, i) = nth(p'_2, i))
{-2}  n' ≥ 0
{-3}  length(p'_1) ≥ 1
{-4}  path?(g')(p'_1)
{-5}  length(p'_2) ≥ 1
{-6}  path?(g')(p'_2)
{-7}  car(p'_1) = car(p'_2)
{-8}  n' < length(p'_1)
{-9}  n' < length(p'_2)
--------------------------------------------------------------------------------
{1}   nd ≥ 1 ∧ path?(g')(head[T](p'_1, nd))
{2}   nd = 0
{3}   nth(p'_1, n') = nth(p'_2, n')
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.1.2.6`.
`path_factor.2`:

```
{-1}  n' ≥ 0
{-2}  length(p'_1) ≥ 1
{-3}  path?(g')(p'_1)
{-4}  length(p'_2) ≥ 1
{-5}  path?(g')(p'_2)
{-6}  car(p'_1) = car(p'_2)
{-7}  n' < length(p'_1)
{-8}  n' < length(p'_2)
--------------------------------------------------------------------------------
{1}   nonempty?[nat]
          (λ (i: nat):
               i < length[T](p'_1) ∧
                 i < length[T](p'_2) ∧ ¬ nth[T](p'_1, i) = nth[T](p'_2, i))
{2}   nth(p'_1, n') = nth(p'_2, n')
{3}   ∃ (q, p1t, p2t: (path?(g'))):
        nth(q, length(q) − 1) = car(p1t) ∧
          nth(q, length(q) − 1) = car(p2t) ∧
            length(p1t) ≥ 2 ∧
              length(p2t) ≥ 2 ∧
                p'_1 = append(q, cdr(p1t)) ∧
                  p'_2 = append(q, cdr(p2t)) ∧
                    nth(p1t, 0) = nth(p2t, 0) ∧ ¬ nth(p1t, 1) = nth(p2t, 1)
```

Hiding formulas: 3,
Installing automatic rewrites from: nonempty? empty? member
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_factor.2`.
Q.E.D.

C.97.33 Graph.node_pair_distinction_TCC1

Terse proof for `node_pair_distinction_TCC1`.

`node_pair_distinction_TCC1:`

{1} ∀ (g: graph_type, P: PRED[[(g‘nodes), (g‘nodes)]]):
(∀ (n_1, n_2: (g‘nodes), p_1, p_2: (path?(g))):
¬ $n_1 = n_2$ ∧
root_path(g, n_1)(p_1) ∧
root_path(g, n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P(n_1, n_2)$)
∧
(∀ (n_1, n_2: (g‘nodes), p: (path?(g))):
¬ $n_1 = n_2$ ∧ path_start(p) = n_1 ∧ path_end(p) = n_2 ∧ length(p) ≥ 2 ⊃
$P(n_1, n_2)$)
∧
(∀ (n_1, n_2: (g‘nodes), p: (path?(g))):
¬ $n_1 = n_2$ ∧ path_start(p) = n_2 ∧ path_end(p) = n_1 ∧ length(p) ≥ 2 ⊃
$P(n_1, n_2)$)
∧ (∀ (n: (g‘nodes)): $P(n, n)$) ∧ finite_tree?(g)
⊃
(∀ (n_{11}, n_{21}, n_3: (g‘nodes), p_{11}, p_{21}: (path?(g))):
g‘edges(n_3, path_start(p_{11})) ∧ ¬ $n_{21} = n_3$ ∧ ¬ $n_{11} = n_3$ ∧ ¬ $n_{11} = n_{21}$ ⊃
g‘nodes(path_start[T](p_{21})))

Repeatedly Skolemizing and flattening,

Rewriting using nodes_path_start, matching in *,

This completes the proof of `node_pair_distinction_TCC1`.

Q.E.D.

C.97.34 Graph.node_pair_distinction

Terse proof for `node_pair_distinction`.

`node_pair_distinction:`

{1} ∀ (g: graph_type, P: PRED[[(g`nodes), ($g$`nodes)]]):
(finite_tree?(g) ∧
(∀ (n: (g`nodes)): $P(n, n)$) ∧
(∀ (n_1, n_2: (g`nodes), p: (path?(g))):
¬ $n_1 = n_2$ ∧ path_start(p) = n_2 ∧ path_end(p) = n_1 ∧ length(p) ≥ 2 ⊃
$P(n_1, n_2)$)
∧
(∀ (n_1, n_2: (g`nodes), p: (path?(g))):
¬ $n_1 = n_2$ ∧ path_start(p) = n_1 ∧ path_end(p) = n_2 ∧ length(p) ≥ 2 ⊃
$P(n_1, n_2)$)
∧
(∀ (n_1, n_2: (g`nodes), p_1, p_2: (path?(g))):
¬ $n_1 = n_2$ ∧
root_path(g, n_1)(p_1) ∧
root_path(g, n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P(n_1, n_2)$)
∧
(∀ (n_1, n_2, n_3: (g`nodes), p_1, p_2: (path?(g))):
¬ $n_1 = n_2$ ∧
¬ $n_1 = n_3$ ∧
¬ $n_2 = n_3$ ∧
g`edges(n_3, path_start(p_1)) ∧
g`edges(n_3, path_start(p_2)) ∧
path_end(p_1) = n_1 ∧
path_end(p_2) = n_2 ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P(n_1, n_2)$)))
⊃ (∀ (n_1, n_2: (g`nodes)): $P(n_1, n_2)$)

Repeatedly Skolemizing and flattening,

Case splitting on n1!1 = n2!1,

we get 2 subgoals:

`node_pair_distinction.1`:

{-1} $n'_1 = n'_2$
{-2} g'`nodes`(n'_1)
{-3} g'`nodes`(n'_2)
{-4} finite_tree?(g')
{-5} $\forall\ (n\colon (g'\text{`nodes}))\colon P'(n,\ n)$
{-6} $\forall\ (n_1,\ n_2\colon (g'\text{`nodes}),\ p\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge \text{path\_start}(p) = n_2 \wedge \text{path\_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset$
$P'(n_1,\ n_2)$
{-7} $\forall\ (n_1,\ n_2\colon (g'\text{`nodes}),\ p\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge \text{path\_start}(p) = n_1 \wedge \text{path\_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$
$P'(n_1,\ n_2)$
{-8} $\forall\ (n_1,\ n_2\colon (g'\text{`nodes}),\ p_1,\ p_2\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge$
$\text{root\_path}(g',\ n_1)(p_1) \wedge$
$\text{root\_path}(g',\ n_2)(p_2) \wedge \neg\ \text{path\_start}(p_1) = \text{path\_start}(p_2)$
$\supset\ P'(n_1,\ n_2)$
{-9} $\forall\ (n_1,\ n_2,\ n_3\colon (g'\text{`nodes}),\ p_1,\ p_2\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge$
$\neg\ n_1 = n_3 \wedge$
$\neg\ n_2 = n_3 \wedge$
$g'\text{`edges}(n_3,\ \text{path\_start}(p_1)) \wedge$
$g'\text{`edges}(n_3,\ \text{path\_start}(p_2)) \wedge$
$\text{path\_end}(p_1) = n_1 \wedge \text{path\_end}(p_2) = n_2 \wedge \neg\ \text{path\_start}(p_1) = \text{path\_start}(p_2)$
$\supset\ P'(n_1,\ n_2)$

{1} $P'(n'_1,\ n'_2)$

Instantiating the top quantifier in -5 with the terms: n'_1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.1`.

`node_pair_distinction.2:`

{-1} g'`‘nodes`(n'_1)
{-2} g'`‘nodes`(n'_2)
{-3} finite_tree?(g')
{-4} $\forall\ (n\colon (g'$`‘nodes`$))\colon P'(n,\ n)$
{-5} $\forall\ (n_1,\ n_2\colon (g'$`‘nodes`$),\ p\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge \text{path\_start}(p) = n_2 \wedge \text{path\_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset$
$P'(n_1,\ n_2)$
{-6} $\forall\ (n_1,\ n_2\colon (g'$`‘nodes`$),\ p\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge \text{path\_start}(p) = n_1 \wedge \text{path\_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset$
$P'(n_1,\ n_2)$
{-7} $\forall\ (n_1,\ n_2\colon (g'$`‘nodes`$),\ p_1,\ p_2\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge$
$\text{root\_path}(g',\ n_1)(p_1) \wedge$
$\text{root\_path}(g',\ n_2)(p_2) \wedge \neg\ \text{path\_start}(p_1) = \text{path\_start}(p_2)$
$\supset\ P'(n_1,\ n_2)$
{-8} $\forall\ (n_1,\ n_2,\ n_3\colon (g'$`‘nodes`$),\ p_1,\ p_2\colon (\text{path?}(g'))):$
$\neg\ n_1 = n_2 \wedge$
$\neg\ n_1 = n_3 \wedge$
$\neg\ n_2 = n_3 \wedge$
g'`‘edges`$(n_3,\ \text{path\_start}(p_1)) \wedge$
g'`‘edges`$(n_3,\ \text{path\_start}(p_2)) \wedge$
$\text{path\_end}(p_1) = n_1 \wedge \text{path\_end}(p_2) = n_2 \wedge \neg\ \text{path\_start}(p_1) = \text{path\_start}(p_2)$
$\supset\ P'(n_1,\ n_2)$

{1} $n'_1 = n'_2$
{2} $P'(n'_1,\ n'_2)$

Hiding formulas: -4,

Simplifying, rewriting, and recording with decision procedures,

Letting rp1 name path_to_root(g!1, n1!1),

Letting rp2 name path_to_root(g!1, n2!1),

Case splitting on EXISTS (i: below(length(rp1))): nth(rp1, i) = n2!1,

we get 2 subgoals:

`node_pair_distinction.2.1:`

{-1} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{-2} path_to_root(g', n_2') = rp2
{-3} path_to_root(g', n_1') = rp1
{-4} g'`nodes(n_1')
{-5} g'`nodes(n_2')
{-6} finite_tree?(g')
{-7} $\forall$ (n_1, n_2: (g'`nodes), p: (path?(g'))):
$\neg$ $n_1 = n_2$ $\wedge$ path_start(p) = n_2 $\wedge$ path_end(p) = n_1 $\wedge$ length(p) $\geq$ 2 $\supset$
$P'(n_1, n_2)$
{-8} $\forall$ (n_1, n_2: (g'`nodes), p: (path?(g'))):
$\neg$ $n_1 = n_2$ $\wedge$ path_start(p) = n_1 $\wedge$ path_end(p) = n_2 $\wedge$ length(p) $\geq$ 2 $\supset$
$P'(n_1, n_2)$
{-9} $\forall$ (n_1, n_2: (g'`nodes), p_1, p_2: (path?(g'))):
$\neg$ $n_1 = n_2$ $\wedge$
root_path(g', n_1)(p_1) $\wedge$
root_path(g', n_2)(p_2) $\wedge$ $\neg$ path_start(p_1) = path_start(p_2)
$\supset$ $P'(n_1, n_2)$
{-10} $\forall$ (n_1, n_2, n_3: (g'`nodes), p_1, p_2: (path?(g'))):
$\neg$ $n_1 = n_2$ $\wedge$
$\neg$ $n_1 = n_3$ $\wedge$
$\neg$ $n_2 = n_3$ $\wedge$
g'`edges(n_3, path_start(p_1)) $\wedge$
g'`edges(n_3, path_start(p_2)) $\wedge$
path_end(p_1) = n_1 $\wedge$ path_end(p_2) = n_2 $\wedge$ $\neg$ path_start(p_1) = path_start(p_2)
$\supset$ $P'(n_1, n_2)$

{1} $n_1' = n_2'$
{2} $P'(n_1', n_2')$

Repeatedly Skolemizing and flattening,

Adding type constraints for rp1,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2, -3,

Installing automatic rewrites from: path_start path_end path_tail nth_tail car_tail length_tail path_list?

Instantiating the top quantifier in -10 with the terms: n_1', n_2', tail(rp1, i'),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.1`.

`node_pair_distinction.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{path\_to\_root}(g',\ n_2') = \text{rp2} \\
\{\text{-}2\} & \text{path\_to\_root}(g',\ n_1') = \text{rp1} \\
\{\text{-}3\} & g'\text{`nodes}(n_1') \\
\{\text{-}4\} & g'\text{`nodes}(n_2') \\
\{\text{-}5\} & \text{finite\_tree?}(g') \\
\{\text{-}6\} & \forall\ (n_1,\ n_2\colon\ (g'\text{`nodes}),\ p\colon\ (\text{path?}(g'))) \colon \\
 & \quad \neg\ n_1 = n_2 \wedge \text{path\_start}(p) = n_2 \wedge \text{path\_end}(p) = n_1 \wedge \text{length}(p) \geq 2 \supset \\
 & \quad\ P'(n_1,\ n_2) \\
\{\text{-}7\} & \forall\ (n_1,\ n_2\colon\ (g'\text{`nodes}),\ p\colon\ (\text{path?}(g'))) \colon \\
 & \quad \neg\ n_1 = n_2 \wedge \text{path\_start}(p) = n_1 \wedge \text{path\_end}(p) = n_2 \wedge \text{length}(p) \geq 2 \supset \\
 & \quad\ P'(n_1,\ n_2) \\
\{\text{-}8\} & \forall\ (n_1,\ n_2\colon\ (g'\text{`nodes}),\ p_1,\ p_2\colon\ (\text{path?}(g'))) \colon \\
 & \quad \neg\ n_1 = n_2 \wedge \\
 & \quad\ \text{root\_path}(g',\ n_1)(p_1) \wedge \\
 & \quad\ \text{root\_path}(g',\ n_2)(p_2) \wedge \neg\ \text{path\_start}(p_1) = \text{path\_start}(p_2) \\
 & \quad \supset\ P'(n_1,\ n_2) \\
\{\text{-}9\} & \forall\ (n_1,\ n_2,\ n_3\colon\ (g'\text{`nodes}),\ p_1,\ p_2\colon\ (\text{path?}(g'))) \colon \\
 & \quad \neg\ n_1 = n_2 \wedge \\
 & \quad\ \neg\ n_1 = n_3 \wedge \\
 & \quad\ \neg\ n_2 = n_3 \wedge \\
 & \quad\ g'\text{`edges}(n_3,\ \text{path\_start}(p_1)) \wedge \\
 & \quad\ g'\text{`edges}(n_3,\ \text{path\_start}(p_2)) \wedge \\
 & \quad\ \text{path\_end}(p_1) = n_1 \wedge \text{path\_end}(p_2) = n_2 \wedge \neg\ \text{path\_start}(p_1) = \text{path\_start}(p_2) \\
 & \quad \supset\ P'(n_1,\ n_2) \\
\hline
\{1\} & \exists\ (i\colon\ \text{below}(\text{length}(\text{rp1}))) \colon\ \text{nth}(\text{rp1},\ i) = n_2' \\
\{2\} & n_1' = n_2' \\
\{3\} & P'(n_1',\ n_2')
\end{array}
$$

Hiding formulas: -6,

Case splitting on EXISTS (i: below(length(rp2))): nth(rp2, i) = n1!1,

we get 2 subgoals:

`node_pair_distinction.2.2.1:`

{-1} $\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{-2} path_to_root(g', n_2') = rp2
{-3} path_to_root(g', n_1') = rp1
{-4} g'`nodes(n_1')
{-5} g'`nodes(n_2')
{-6} finite_tree?(g')
{-7} $\forall$ (n_1, n_2: (g'`nodes), p: (path?(g'))):
¬ $n_1 = n_2 \wedge$ path_start(p) = $n_1 \wedge$ path_end(p) = $n_2 \wedge$ length(p) $\geq 2 \supset$
$P'(n_1, n_2)$
{-8} $\forall$ (n_1, n_2: (g'`nodes), p_1, p_2: (path?(g'))):
¬ $n_1 = n_2 \wedge$
root_path(g', n_1)(p_1) $\wedge$
root_path(g', n_2)(p_2) $\wedge$ ¬ path_start(p_1) = path_start(p_2)
$\supset P'(n_1, n_2)$
{-9} $\forall$ (n_1, n_2, n_3: (g'`nodes), p_1, p_2: (path?(g'))):
¬ $n_1 = n_2 \wedge$
¬ $n_1 = n_3 \wedge$
¬ $n_2 = n_3 \wedge$
g'`edges(n_3, path_start(p_1)) $\wedge$
g'`edges(n_3, path_start(p_2)) $\wedge$
path_end(p_1) = $n_1 \wedge$ path_end(p_2) = $n_2 \wedge$ ¬ path_start(p_1) = path_start(p_2)
$\supset P'(n_1, n_2)$

{1} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{2} $n_1' = n_2'$
{3} $P'(n_1', n_2')$

Repeatedly Skolemizing and flattening,

Adding type constraints for rp2,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2, -3,

Installing automatic rewrites from: path_start path_end path_tail nth_tail car_tail length_tail path_list?

Instantiating the top quantifier in -10 with the terms: n_1', n_2', tail(rp2, i'),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.1`.

`node_pair_distinction.2.2.2:`

{-1}	path_to_root(g', n_2') = rp2
{-2}	path_to_root(g', n_1') = rp1
{-3}	g'‘nodes(n_1')
{-4}	g'‘nodes(n_2')
{-5}	finite_tree?(g')
{-6}	∀ (n_1, n_2: (g'‘nodes), p: (path?(g'))): ¬ $n_1 = n_2$ ∧ path_start(p) = n_1 ∧ path_end(p) = n_2 ∧ length(p) ≥ 2 ⊃ $P'(n_1, n_2)$
{-7}	∀ (n_1, n_2: (g'‘nodes), p_1, p_2: (path?(g'))): ¬ $n_1 = n_2$ ∧ root_path(g', n_1)(p_1) ∧ root_path(g', n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2) ⊃ $P'(n_1, n_2)$
{-8}	∀ (n_1, n_2, n_3: (g'‘nodes), p_1, p_2: (path?(g'))): ¬ $n_1 = n_2$ ∧ ¬ $n_1 = n_3$ ∧ ¬ $n_2 = n_3$ ∧ g'‘edges(n_3, path_start(p_1)) ∧ g'‘edges(n_3, path_start(p_2)) ∧ path_end(p_1) = n_1 ∧ path_end(p_2) = n_2 ∧ ¬ path_start(p_1) = path_start(p_2) ⊃ $P'(n_1, n_2)$
{1}	∃ (i: below(length(rp2))): nth(rp2, i) = n_1'
{2}	∃ (i: below(length(rp1))): nth(rp1, i) = n_2'
{3}	$n_1' = n_2'$
{4}	$P'(n_1', n_2')$

Hiding formulas: -6,

Case splitting on path_start(rp1) = path_start(rp2),

we get 2 subgoals:

node_pair_distinction.2.2.2.1:

{-1} path_start(rp1) = path_start(rp2)
{-2} path_to_root(g', n_2') = rp2
{-3} path_to_root(g', n_1') = rp1
{-4} g'`nodes(n_1')
{-5} g'`nodes(n_2')
{-6} finite_tree?(g')
{-7} $\forall$ (n_1, n_2: (g'`nodes), p_1, p_2: (path?(g'))):
¬ $n_1 = n_2$ ∧
root_path(g', n_1)(p_1) ∧
root_path(g', n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P'(n_1, n_2)$
{-8} $\forall$ (n_1, n_2, n_3: (g'`nodes), p_1, p_2: (path?(g'))):
¬ $n_1 = n_2$ ∧
¬ $n_1 = n_3$ ∧
¬ $n_2 = n_3$ ∧
g'`edges(n_3, path_start(p_1)) ∧
g'`edges(n_3, path_start(p_2)) ∧
path_end(p_1) = n_1 ∧ path_end(p_2) = n_2 ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P'(n_1, n_2)$

{1} $\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{2} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{3} $n_1' = n_2'$
{4} $P'(n_1', n_2')$

Hiding formulas: -7,

Installing automatic rewrites from: path_list? length_cdr path_start nth path_end concat_path nth_append_right length_append nth_append_left

Using lemma path_factor,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`node_pair_distinction.2.2.2.1.1:`

{-1} car(rp1) = car(rp2)
{-2} ∃ (q, p1t, p2t: (path?(g'))):
concatable?(q, p1t) ∧
concatable?(q, p2t) ∧
length(p1t) ≥ 2 ∧
length(p2t) ≥ 2 ∧
rp1 = append(q, cdr(p1t)) ∧
rp2 = append(q, cdr(p2t)) ∧
car(p1t) = car(p2t) ∧ ¬ car(cdr(p1t)) = car(cdr(p2t))
{-3} path_to_root(g', n_2') = rp2
{-4} path_to_root(g', n_1') = rp1
{-5} g'`nodes(n_1')
{-6} g'`nodes(n_2')
{-7} finite_tree?(g')
{-8} ∀ (n_1, n_2, n_3: (g'`nodes), p_1, p_2: (path?(g'))):
¬ n_1 = n_2 ∧
¬ n_1 = n_3 ∧
¬ n_2 = n_3 ∧
g'`edges(n_3, car(p_1)) ∧
g'`edges(n_3, car(p_2)) ∧
nth(p_1, length(p_1) − 1) = n_1 ∧
nth(p_2, length(p_2) − 1) = n_2 ∧ ¬ car(p_1) = car(p_2)
⊃ P'(n_1, n_2)

{1} ∃ (i: below(length(rp2))): nth(rp2, i) = n_1'
{2} ∃ (i: below(length(rp1))): nth(rp1, i) = n_2'
{3} n_1' = n_2'
{4} P'(n_1', n_2')

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -3, -5,

Using lemma path_cdr,

Using lemma path_cdr,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -20 with the terms: n_1', n_2', car(p1t′), cdr(p1t′), cdr(p2t′),

we get 2 subgoals:

`node_pair_distinction.2.2.2.1.1.1:`

{-1} path?(g')(cdr(p2t$'$))
{-2} path?(g')(cdr(p1t$'$))
{-3} length(q') $\geq$ 1
{-4} path?(g')(q')
{-5} path?(g')(p1t$'$)
{-6} path?(g')(p2t$'$)
{-7} car(rp1) = car(rp2)
{-8} concatable?(q', p1t$'$)
{-9} concatable?(q', p2t$'$)
{-10} length(p1t$'$) $\geq$ 2
{-11} length(p2t$'$) $\geq$ 2
{-12} rp1 = append(q', cdr(p1t$'$))
{-13} rp2 = append(q', cdr(p2t$'$))
{-14} car(p1t$'$) = car(p2t$'$)
{-15} path_to_root(g', n'_2) = rp2
{-16} path_to_root(g', n'_1) = rp1
{-17} g'`nodes(n'_1)
{-18} g'`nodes(n'_2)
{-19} finite_tree?(g')
{-20} $\neg$ $n'_1 = n'_2$ $\wedge$
$\neg$ n'_1 = car(p1t$'$) $\wedge$
$\neg$ n'_2 = car(p1t$'$) $\wedge$
g'`edges(car(p1t$'$), car(cdr(p1t$'$))) $\wedge$
g'`edges(car(p1t$'$), car(cdr(p2t$'$))) $\wedge$
nth(cdr(p1t$'$), length(cdr(p1t$'$)) $-$ 1) = n'_1 $\wedge$
nth(cdr(p2t$'$), length(cdr(p2t$'$)) $-$ 1) = n'_2 $\wedge$
$\neg$ car(cdr(p1t$'$)) = car(cdr(p2t$'$))
$\supset$ $P'(n'_1, n'_2)$

{1} car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2} $\exists$ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4} $n'_1 = n'_2$
{5} $P'(n'_1, n'_2)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 6 subgoals:

`node_pair_distinction.2.2.2.1.1.1.1:`

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n_2') = rp2
{-16}	path_to_root(g', n_1') = rp1
{-17}	g'`nodes(n_1')
{-18}	g'`nodes(n_2')
{-19}	finite_tree?(g')
{-20}	n_1' = car(p1t$'$)
{1}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{3}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{4}	n_1' = n_2'
{5}	$P'(n_1', n_2')$

Keeping (-4 -6 -9 -11 -13 -14 -20 2) and hiding *,

Expanding the definition of concatable?,

Replacing using formula -5,

Instantiating the top quantifier in 1 with the terms: length(q') $-$ 1,

we get 2 subgoals:

`node_pair_distinction.2.2.2.1.1.1.1.1:`

{-1}	path?(g')(q')
{-2}	path?(g')(p2t$'$)
{-3}	path_end(q') = path_start(p2t$'$)
{-4}	length(p2t$'$) $\geq$ 2
{-5}	rp2 = append(q', cdr(p2t$'$))
{-6}	car(p1t$'$) = car(p2t$'$)
{-7}	n_1' = car(p1t$'$)
{1}	nth(append(q', cdr(p2t$'$)), length(q') $-$ 1) = n_1'

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.1.1`.

`node_pair_distinction.2.2.2.1.1.1.1.2:`

{-1}	path?(g')(q')
{-2}	path?(g')(p2t$'$)
{-3}	path_end(q') = path_start(p2t$'$)
{-4}	length(p2t$'$) $\geq$ 2
{-5}	rp2 = append(q', cdr(p2t$'$))
{-6}	car(p1t$'$) = car(p2t$'$)
{-7}	n_1' = car(p1t$'$)
{1}	length$\lceil T \rceil$(q') $-$ 1 $<$ length$\lceil T \rceil$(rp2)

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.1.2`.

`node_pair_distinction.2.2.2.1.1.1.2:`

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n_2') = rp2
{-16}	path_to_root(g', n_1') = rp1
{-17}	g'`nodes(n_1')
{-18}	g'`nodes(n_2')
{-19}	finite_tree?(g')
{-20}	n_2' = car(p1t$'$)
{1}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{3}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{4}	n_1' = n_2'
{5}	P'(n_1', n_2')

Keeping (-4 -5 -8 -10 -12 -20 3) and hiding *,

Expanding the definition of concatable?,

Replacing using formula -5,

Instantiating the top quantifier in 1 with the terms: length(q') $-$ 1,

we get 2 subgoals:

`node_pair_distinction.2.2.2.1.1.1.2.1:`

{-1}	path?(g')(q')
{-2}	path?(g')(p1t$'$)
{-3}	path_end(q') = path_start(p1t$'$)
{-4}	length(p1t$'$) $\geq$ 2
{-5}	rp1 = append(q', cdr(p1t$'$))
{-6}	n_2' = car(p1t$'$)
{1}	nth(append(q', cdr(p1t$'$)), length(q') $-$ 1) = n_2'

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `node_pair_distinction.2.2.2.1.1.1.2.1`.
`node_pair_distinction.2.2.2.1.1.1.2.2:`

{-1}	path?(g')(q')
{-2}	path?(g')(p1t$'$)
{-3}	path_end(q') = path_start(p1t$'$)
{-4}	length(p1t$'$) $\geq$ 2
{-5}	rp1 = append(q', cdr(p1t$'$))
{-6}	n_2' = car(p1t$'$)
{1}	length$\lceil T \rceil$(q') $-$ 1 $<$ length$\lceil T \rceil$(rp1)

Replacing using formula -5,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `node_pair_distinction.2.2.2.1.1.1.2.2`.
`node_pair_distinction.2.2.2.1.1.1.3:`

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n_2') = rp2
{-16}	path_to_root(g', n_1') = rp1
{-17}	g'`nodes(n_1')
{-18}	g'`nodes(n_2')
{-19}	finite_tree?(g')
{1}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{3}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{4}	n_1' = n_2'
{5}	nth(cdr(p2t$'$), length$\lceil T \rceil$(p2t$'$) $-$ 2) = n_2'
{6}	P'(n_1', n_2')

Keeping (-4 -6 -9 -13 -15 5) and hiding *,
Adding type constraints for rp2,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -1, -2, -3, -9,

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.3`.

`node_pair_distinction.2.2.2.1.1.1.4`:

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n_2') = rp2
{-16}	path_to_root(g', n_1') = rp1
{-17}	g'`nodes(n_1')
{-18}	g'`nodes(n_2')
{-19}	finite_tree?(g')
{1}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{3}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{4}	$n_1' = n_2'$
{5}	nth(cdr(p1t$'$), length$[T]$(p1t$'$) $-$ 2) = n_1'
{6}	$P'(n_1', n_2')$

Keeping (-4 -5 -8 -12 -16 5) and hiding *,

Adding type constraints for rp1,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -1, -2, -3, -9,

Replacing using formula -5,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.4`.

`node_pair_distinction.2.2.2.1.1.1.5:`

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n'_2) = rp2
{-16}	path_to_root(g', n'_1) = rp1
{-17}	g'`nodes(n'_1)
{-18}	g'`nodes(n'_2)
{-19}	finite_tree?(g')
{1}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4}	$n'_1 = n'_2$
{5}	g'`edges(car(p1t$'$), car(cdr(p2t$'$)))
{6}	$P'(n'_1, n'_2)$

Keeping (-6 -14 5) and hiding *,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -2 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.5`.

`node_pair_distinction.2.2.2.1.1.1.6:`

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n_2') = rp2
{-16}	path_to_root(g', n_1') = rp1
{-17}	g'`nodes(n_1')
{-18}	g'`nodes(n_2')
{-19}	finite_tree?(g')
{1}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{2}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n_1'
{3}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n_2'
{4}	n_1' = n_2'
{5}	g'`edges(car(p1t$'$), car(cdr(p1t$'$)))
{6}	P'(n_1', n_2')

Keeping (-5 5) and hiding *,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -2 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.1.6`.

`node_pair_distinction.2.2.2.1.1.2:`

{-1}	path?(g')(cdr(p2t$'$))
{-2}	path?(g')(cdr(p1t$'$))
{-3}	length(q') $\geq$ 1
{-4}	path?(g')(q')
{-5}	path?(g')(p1t$'$)
{-6}	path?(g')(p2t$'$)
{-7}	car(rp1) = car(rp2)
{-8}	concatable?(q', p1t$'$)
{-9}	concatable?(q', p2t$'$)
{-10}	length(p1t$'$) $\geq$ 2
{-11}	length(p2t$'$) $\geq$ 2
{-12}	rp1 = append(q', cdr(p1t$'$))
{-13}	rp2 = append(q', cdr(p2t$'$))
{-14}	car(p1t$'$) = car(p2t$'$)
{-15}	path_to_root(g', n'_2) = rp2
{-16}	path_to_root(g', n'_1) = rp1
{-17}	g'`nodes(n'_1)
{-18}	g'`nodes(n'_2)
{-19}	finite_tree?(g')
{1}	g'`nodes(car$\lfloor T \rceil$(p1t$'$))
{2}	car(cdr(p1t$'$)) = car(cdr(p2t$'$))
{3}	$\exists$ (i: below(length(rp2))): nth(rp2, i) = n'_1
{4}	$\exists$ (i: below(length(rp1))): nth(rp1, i) = n'_2
{5}	$n'_1 = n'_2$
{6}	$P'(n'_1, n'_2)$

Keeping (-5 1) and hiding *,

Expanding the definition of path?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -1 with the terms: 0,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.1.2`.

`node_pair_distinction.2.2.2.1.2:`

{-1} car(rp1) = car(rp2)
{-2} path_to_root(g', n'_2) = rp2
{-3} path_to_root(g', n'_1) = rp1
{-4} g'`nodes(n'_1)
{-5} g'`nodes(n'_2)
{-6} finite_tree?(g')
{-7} $\forall$ (n_1, n_2, n_3: (g'`nodes), p_1, p_2: (path?(g'))):
$\neg\ n_1 = n_2 \wedge$
$\neg\ n_1 = n_3 \wedge$
$\neg\ n_2 = n_3 \wedge$
g'`edges(n_3, car(p_1)) $\wedge$
g'`edges(n_3, car(p_2)) $\wedge$
nth(p_1, length(p_1) $-$ 1) = $n_1 \wedge$
nth(p_2, length(p_2) $-$ 1) = $n_2 \wedge \neg$ car(p_1) = car(p_2)
$\supset\ P'(n_1, n_2)$

{1} $\exists$ (n: nat): $n <$ length(rp1) $\wedge\ n <$ length(rp2) $\wedge\ \neg$ nth(rp1, n) = nth(rp2, n)
{2} $\exists$ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4} $n'_1 = n'_2$
{5} $P'(n'_1, n'_2)$

Hiding formulas: -1, -4, -5, -6, -7, 5,
Case splitting on length(rp1) < length(rp2),
we get 2 subgoals:
`node_pair_distinction.2.2.2.1.2.1:`

{-1} length(rp1) < length(rp2)
{-2} path_to_root(g', n'_2) = rp2
{-3} path_to_root(g', n'_1) = rp1

{1} $\exists$ (n: nat): $n <$ length(rp1) $\wedge\ n <$ length(rp2) $\wedge\ \neg$ nth(rp1, n) = nth(rp2, n)
{2} $\exists$ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4} $n'_1 = n'_2$

Adding type constraints for rp1,
Expanding the definition of root_path,
Applying disjunctive simplification to flatten sequent,
Hiding formulas: -2, -3,
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in 1 with the terms: length(rp1) $-$ 1,
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in 1 with the terms: length(rp1) $-$ 1,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `node_pair_distinction.2.2.2.1.2.1`.
`node_pair_distinction.2.2.2.1.2.2:`

{-1} path_to_root(g', n'_2) = rp2
{-2} path_to_root(g', n'_1) = rp1

{1} length(rp1) < length(rp2)
{2} $\exists$ (n: nat): $n <$ length(rp1) $\wedge\ n <$ length(rp2) $\wedge\ \neg$ nth(rp1, n) = nth(rp2, n)
{3} $\exists$ (i: below(length(rp2))): nth(rp2, i) = n'_1
{4} $\exists$ (i: below(length(rp1))): nth(rp1, i) = n'_2
{5} $n'_1 = n'_2$

Adding type constraints for rp2,

Expanding the definition of root_path,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: -2, -3,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in 2 with the terms: length(rp2) − 1,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in 3 with the terms: length(rp2) − 1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.1.2.2`.

`node_pair_distinction.2.2.2.2`:

```
{-1}  path_to_root(g', n'_2) = rp2
{-2}  path_to_root(g', n'_1) = rp1
{-3}  g'`nodes(n'_1)
{-4}  g'`nodes(n'_2)
{-5}  finite_tree?(g')
{-6}  ∀ (n_1, n_2: (g'`nodes), p_1, p_2: (path?(g'))):
        ¬ n_1 = n_2 ∧
         root_path(g', n_1)(p_1) ∧
          root_path(g', n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2)
         ⊃ P'(n_1, n_2)
{-7}  ∀ (n_1, n_2, n_3: (g'`nodes), p_1, p_2: (path?(g'))):
        ¬ n_1 = n_2 ∧
         ¬ n_1 = n_3 ∧
          ¬ n_2 = n_3 ∧
           g'`edges(n_3, path_start(p_1)) ∧
            g'`edges(n_3, path_start(p_2)) ∧
             path_end(p_1) = n_1 ∧ path_end(p_2) = n_2 ∧ ¬ path_start(p_1) = path_start(p_2)
         ⊃ P'(n_1, n_2)
-------------------------------------------------------------------------
{1}  path_start(rp1) = path_start(rp2)
{2}  ∃ (i: below(length(rp2))): nth(rp2, i) = n'_1
{3}  ∃ (i: below(length(rp1))): nth(rp1, i) = n'_2
{4}  n'_1 = n'_2
{5}  P'(n'_1, n'_2)
```

Hiding formulas: -7,

Instantiating the top quantifier in -6 with the terms: n'_1, n'_2, rp1, rp2,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `node_pair_distinction.2.2.2.2`.

Q.E.D.

C.97.35 Graph.symmetric_node_pair_distinction_TCC1

Terse proof for `symmetric_node_pair_distinction_TCC1`.

`symmetric_node_pair_distinction_TCC1`:

{1} ∀ (g: graph_type, P: PRED[[(g‘nodes), (g‘nodes)]]):
($\forall$ (n_1, n_2: (g‘nodes), p_1, p_2: (path?(g))):
¬ $n_1 = n_2$ ∧
root_path(g, n_1)(p_1) ∧
root_path(g, n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P(n_1, n_2)$)
∧
($\forall$ (n_1, n_2: (g‘nodes), p: (path?(g))):
¬ $n_1 = n_2$ ∧ path_start(p) = n_2 ∧ path_end(p) = n_1 ∧ length(p) ≥ 2 ⊃
$P(n_1, n_2)$)
∧ ($\forall$ (n: (g‘nodes)): $P(n, n)$) ∧ symmetric?(P) ∧ finite_tree?(g)
⊃
($\forall$ (n_{11}, n_{21}, n_3: (g‘nodes), p_{11}, p_{21}: (path?(g))):
g‘edges(n_3, path_start(p_{11})) ∧ ¬ $n_{21} = n_3$ ∧ ¬ $n_{11} = n_3$ ∧ ¬ $n_{11} = n_{21}$ ⊃
g‘nodes(path_start[T](p_{21})))

Repeatedly Skolemizing and flattening,
Rewriting using nodes_path_start, matching in *,
This completes the proof of `symmetric_node_pair_distinction_TCC1`.
Q.E.D.

C.97.36 Graph.symmetric_node_pair_distinction

Terse proof for `symmetric_node_pair_distinction`.

`symmetric_node_pair_distinction`:

{1} ∀ (g: graph_type, P: PRED[[(g‘nodes), (g‘nodes)]]):
(finite_tree?(g) ∧
symmetric?(P) ∧
($\forall$ (n: (g‘nodes)): $P(n, n)$) ∧
($\forall$ (n_1, n_2: (g‘nodes), p: (path?(g))):
¬ $n_1 = n_2$ ∧ path_start(p) = n_2 ∧ path_end(p) = n_1 ∧ length(p) ≥ 2 ⊃
$P(n_1, n_2)$)
∧
($\forall$ (n_1, n_2: (g‘nodes), p_1, p_2: (path?(g))):
¬ $n_1 = n_2$ ∧
root_path(g, n_1)(p_1) ∧
root_path(g, n_2)(p_2) ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P(n_1, n_2)$)
∧
($\forall$ (n_1, n_2, n_3: (g‘nodes), p_1, p_2: (path?(g))):
¬ $n_1 = n_2$ ∧
¬ $n_1 = n_3$ ∧
¬ $n_2 = n_3$ ∧
g‘edges(n_3, path_start(p_1)) ∧
g‘edges(n_3, path_start(p_2)) ∧
path_end(p_1) = n_1 ∧
path_end(p_2) = n_2 ∧ ¬ path_start(p_1) = path_start(p_2)
⊃ $P(n_1, n_2)$)))
⊃ ($\forall$ (n_1, n_2: (g‘nodes)): $P(n_1, n_2)$)

Skolemizing and flattening,
Using lemma node_pair_distinction,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: -2, -4, -5, 2,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -6 with the terms: n_2', n_1', p',
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of symmetric?,
Instantiating the top quantifier in -10 with the terms: n_2', n_1',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `symmetric_node_pair_distinction`.
Q.E.D.

C.98 Proofs for Graph_Relations (graph.pvs)

C.98.1 Graph_Relations.trans_closure_char_TCC1

Terse proof for `trans_closure_char_TCC1`.
`trans_closure_char_TCC1`:

$$\{1\}\quad \forall\ (x,\ y\colon T,\ l\colon \mathrm{list}[T])\colon \mathrm{length}(l) \geq 2 \supset \mathrm{path\_list?}[T](l)$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_list? length
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char_TCC1`.
Q.E.D.

C.98.2 Graph_Relations.trans_closure_char_TCC2

Terse proof for `trans_closure_char_TCC2`.
`trans_closure_char_TCC2`:

$$\{1\}\quad \forall\ (x,\ y\colon T,\ l\colon \mathrm{list}[T])\colon \mathrm{path\_end}(l) = y \wedge \mathrm{path\_start}(l) = x \wedge \mathrm{length}(l) \geq 2 \supset (\forall\ (i\colon \mathrm{below}(\mathrm{length}[T](l) - 1))\colon i < \mathrm{length}[T](l))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `trans_closure_char_TCC2`.
Q.E.D.

C.98.3 Graph_Relations.trans_closure_char_TCC3

Terse proof for `trans_closure_char_TCC3`.
`trans_closure_char_TCC3`:

$$\{1\}\quad \forall\ (x,\ y\colon T,\ l\colon \mathrm{list}[T])\colon \mathrm{path\_end}(l) = y \wedge \mathrm{path\_start}(l) = x \wedge \mathrm{length}(l) \geq 2 \supset (\forall\ (i\colon \mathrm{below}(\mathrm{length}[T](l) - 1))\colon i + 1 < \mathrm{length}[T](l))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `trans_closure_char_TCC3`.

Q.E.D.

C.98.4 Graph_Relations.trans_closure_char

Terse proof for `trans_closure_char`.

`trans_closure_char`:

$$
\begin{array}{l}
\\
\hline
\{1\}\quad \forall\ (R\colon\ \mathrm{PRED}\big[\big[T,\ T\big]\big],\ x,\ y\colon\ T)\colon \\
\qquad \mathrm{transitive\_closure}(R)(x,\ y)\ \equiv \\
\qquad\ (\exists\ (l\colon\ \mathrm{list}\big[T\big])\colon \\
\qquad\qquad \mathrm{length}(l)\ \geq\ 2\ \wedge \\
\qquad\qquad\ \mathrm{path\_start}(l)\ =\ x\ \wedge \\
\qquad\qquad\ \mathrm{path\_end}(l)\ =\ y\ \wedge \\
\qquad\qquad\ (\forall\ (i\colon\ \mathrm{below}(\mathrm{length}(l)-1))\colon \\
\qquad\qquad\qquad R(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ i+1))))
\end{array}
$$

Installing automatic rewrites from: path_list? length length_is_cons length_cdr

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`trans_closure_char.1`:

$$
\begin{array}{l}
\{\text{-}1\}\quad \mathrm{transitive\_closure}(R')(x',\ y') \\
\hline
\{1\}\quad \exists\ (l\colon\ \mathrm{list}\big[T\big])\colon \\
\qquad \mathrm{length}(l)\ \geq\ 2\ \wedge \\
\qquad\ \mathrm{path\_start}(l)\ =\ x'\ \wedge \\
\qquad\ \mathrm{path\_end}(l)\ =\ y'\ \wedge \\
\qquad\ (\forall\ (i\colon\ \mathrm{below}(\mathrm{length}(l)-1))\colon\ R'(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ 1+i)))
\end{array}
$$

Expanding the definition of transitive_closure,

Instantiating the top quantifier in -1 with the terms: $\lambda\ (x,\ y\colon\ T)\colon\ \exists\ (l\colon\ \mathrm{list}\big[T\big])\colon\ \mathrm{length}(l)\ \geq\ 2\ \wedge$ path_start$(l) = x \wedge$ path_end$(l) = y \wedge (\forall\ (i\colon$ below$(\mathrm{length}(l)-1))\colon\ R'(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ 1+i)))$,

we get 3 subgoals:

`trans_closure_char.1.1:`

{-1} $(R' \subseteq \lambda\ (x,\ y:\ T):\ \exists\ (l:\ \mathrm{list}[T]):\ \mathrm{length}(l)\ \geq\ 2\ \wedge\ \mathrm{path\_start}(l)\ =\ x\ \wedge\ \mathrm{path\_end}(l)\ =$
$\wedge$
$\mathrm{transitive?}(\lambda\ (x,\ y:\ T):$
$\exists\ (l:\ \mathrm{list}[T]):$
$\mathrm{length}(l)\ \geq\ 2\ \wedge$
$\mathrm{path\_start}(l)\ =\ x\ \wedge$
$\mathrm{path\_end}(l)\ =\ y\ \wedge$
$(\forall\ (i:\ \mathrm{below}(\mathrm{length}(l)-1)):$
$R'(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ 1+i))))$
$\supset$
$(\exists\ (l:\ \mathrm{list}[T]):$
$\mathrm{length}(l)\ \geq\ 2\ \wedge$
$\mathrm{path\_start}(l)\ =\ x'\ \wedge$
$\mathrm{path\_end}(l)\ =\ y'\ \wedge$
$(\forall\ (i:\ \mathrm{below}(\mathrm{length}(l)-1)):$
$R'(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ 1+i))))$

{1} $\exists\ (l:\ \mathrm{list}[T]):$
$\mathrm{length}(l)\ \geq\ 2\ \wedge$
$\mathrm{path\_start}(l)\ =\ x'\ \wedge$
$\mathrm{path\_end}(l)\ =\ y'\ \wedge$
$(\forall\ (i:\ \mathrm{below}(\mathrm{length}(l)-1)):\ R'(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ 1+i)))$

Applying propositional simplification,

we get 2 subgoals:

`trans_closure_char.1.1.1:`

{1} $(R' \subseteq \lambda\ (x,\ y:\ T):\ \exists\ (l:\ \mathrm{list}[T]):\ \mathrm{length}(l)\ \geq\ 2\ \wedge\ \mathrm{path\_start}(l)\ =\ x\ \wedge\ \mathrm{path\_end}(l)\ =$
{2} $\exists\ (l:\ \mathrm{list}[T]):$
$\mathrm{length}(l)\ \geq\ 2\ \wedge$
$\mathrm{path\_start}(l)\ =\ x'\ \wedge$
$\mathrm{path\_end}(l)\ =\ y'\ \wedge$
$(\forall\ (i:\ \mathrm{below}(\mathrm{length}(l)-1)):\ R'(\mathrm{nth}(l,\ i),\ \mathrm{nth}(l,\ 1+i)))$

Hiding formulas: 2,

Expanding the definition(s) of (subset? member),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: (:x''`1, $x''$`2:),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `trans_closure_char.1.1.1`.

`trans_closure_char.1.1.2:`

{1} transitive?(λ (x, y: T):
∃ (l: list[T]):
length(l) ≥ 2 ∧
path_start(l) = x ∧
path_end(l) = y ∧
(∀ (i: below(length(l) − 1)):
R′(nth(l, i), nth(l, 1 + i))))
{2} ∃ (l: list[T]):
length(l) ≥ 2 ∧
path_start(l) = x′ ∧
path_end(l) = y′ ∧
(∀ (i: below(length(l) − 1)): R′(nth(l, i), nth(l, 1 + i)))

Hiding formulas: 2,
Expanding the definition of transitive?,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: length_append nth_append_right nth_append_left
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in 1 with the terms: append(l′, cdr(l″)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`trans_closure_char.1.1.2.1:`

{-1} length(l′) ≥ 2
{-2} path_start(l′) = x″
{-3} path_end(l′) = y″
{-4} ∀ (i: below(length(l′) − 1)): R′(nth(l′, i), nth(l′, 1 + i))
{-5} length(l″) ≥ 2
{-6} path_start(l″) = y″
{-7} path_end(l″) = z′
{-8} ∀ (i: below(length(l″) − 1)): R′(nth(l″, i), nth(l″, 1 + i))

{1} ∀ (i: below(length(append(l′, cdr(l″))) − 1)):
R′(nth(append(l′, cdr(l″)), i), nth(append(l′, cdr(l″)), 1 + i))

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on i!1 < length(l!1) - 1,
we get 2 subgoals:
`trans_closure_char.1.1.2.1.1:`

{-1} i′ < length(l′) − 1
{-2} i′ < length(l′) + length(l″) − 2
{-3} length(l′) ≥ 2
{-4} path_start(l′) = x″
{-5} path_end(l′) = y″
{-6} ∀ (i: below(length(l′) − 1)): R′(nth(l′, i), nth(l′, 1 + i))
{-7} length(l″) ≥ 2
{-8} path_start(l″) = y″
{-9} path_end(l″) = z′
{-10} ∀ (i: below(length(l″) − 1)): R′(nth(l″, i), nth(l″, 1 + i))

{1} R′(nth(append(l′, cdr(l″)), i′), nth(append(l′, cdr(l″)), 1 + i′))

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

This completes the proof of `trans_closure_char.1.1.2.1.1`.

`trans_closure_char.1.1.2.1.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & i' < \text{length}(l') + \text{length}(l'') - 2 \\
\{\text{-}2\} & \text{length}(l') \geq 2 \\
\{\text{-}3\} & \text{path\_start}(l') = x'' \\
\{\text{-}4\} & \text{path\_end}(l') = y'' \\
\{\text{-}5\} & \forall\ (i\colon\ \text{below}(\text{length}(l') - 1))\colon\ R'(\text{nth}(l',\ i),\ \text{nth}(l',\ 1 + i)) \\
\{\text{-}6\} & \text{length}(l'') \geq 2 \\
\{\text{-}7\} & \text{path\_start}(l'') = y'' \\
\{\text{-}8\} & \text{path\_end}(l'') = z' \\
\{\text{-}9\} & \forall\ (i\colon\ \text{below}(\text{length}(l'') - 1))\colon\ R'(\text{nth}(l'',\ i),\ \text{nth}(l'',\ 1 + i)) \\
\hline
\{1\} & i' < \text{length}(l') - 1 \\
\{2\} & R'(\text{nth}(\text{append}(l',\ \text{cdr}(l'')),\ i'),\ \text{nth}(\text{append}(l',\ \text{cdr}(l'')),\ 1 + i'))
\end{array}
$$

Case splitting on i!1 = length(l!1) - 1,

we get 2 subgoals:

`trans_closure_char.1.1.2.1.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & i' = \text{length}(l') - 1 \\
\{\text{-}2\} & i' < \text{length}(l') + \text{length}(l'') - 2 \\
\{\text{-}3\} & \text{length}(l') \geq 2 \\
\{\text{-}4\} & \text{path\_start}(l') = x'' \\
\{\text{-}5\} & \text{path\_end}(l') = y'' \\
\{\text{-}6\} & \forall\ (i\colon\ \text{below}(\text{length}(l') - 1))\colon\ R'(\text{nth}(l',\ i),\ \text{nth}(l',\ 1 + i)) \\
\{\text{-}7\} & \text{length}(l'') \geq 2 \\
\{\text{-}8\} & \text{path\_start}(l'') = y'' \\
\{\text{-}9\} & \text{path\_end}(l'') = z' \\
\{\text{-}10\} & \forall\ (i\colon\ \text{below}(\text{length}(l'') - 1))\colon\ R'(\text{nth}(l'',\ i),\ \text{nth}(l'',\ 1 + i)) \\
\hline
\{1\} & i' < \text{length}(l') - 1 \\
\{2\} & R'(\text{nth}(\text{append}(l',\ \text{cdr}(l'')),\ i'),\ \text{nth}(\text{append}(l',\ \text{cdr}(l'')),\ 1 + i'))
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of path_end,

Replacing using formula -1,

Replacing using formula -5,

Instantiating the top quantifier in -10 with the terms: 0,

Expanding the definition of nth,

Expanding the definition of path_start,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `trans_closure_char.1.1.2.1.2.1`.

`trans_closure_char.1.1.2.1.2.2:`

{-1}	$i' < \text{length}(l') + \text{length}(l'') - 2$
{-2}	$\text{length}(l') \geq 2$
{-3}	$\text{path\_start}(l') = x''$
{-4}	$\text{path\_end}(l') = y''$
{-5}	$\forall\ (i\colon\ \text{below}(\text{length}(l') - 1))\colon\ R'(\text{nth}(l',\ i),\ \text{nth}(l',\ 1 + i))$
{-6}	$\text{length}(l'') \geq 2$
{-7}	$\text{path\_start}(l'') = y''$
{-8}	$\text{path\_end}(l'') = z'$
{-9}	$\forall\ (i\colon\ \text{below}(\text{length}(l'') - 1))\colon\ R'(\text{nth}(l'',\ i),\ \text{nth}(l'',\ 1 + i))$
{1}	$i' = \text{length}(l') - 1$
{2}	$i' < \text{length}(l') - 1$
{3}	$R'(\text{nth}(\text{append}(l',\ \text{cdr}(l'')),\ i'),\ \text{nth}(\text{append}(l',\ \text{cdr}(l'')),\ 1 + i'))$

Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in -9 with the terms: $i' - \text{length}(l') + 1$,
Expanding the definition of nth,
which is trivially true.
This completes the proof of `trans_closure_char.1.1.2.1.2.2`.
`trans_closure_char.1.1.2.2:`

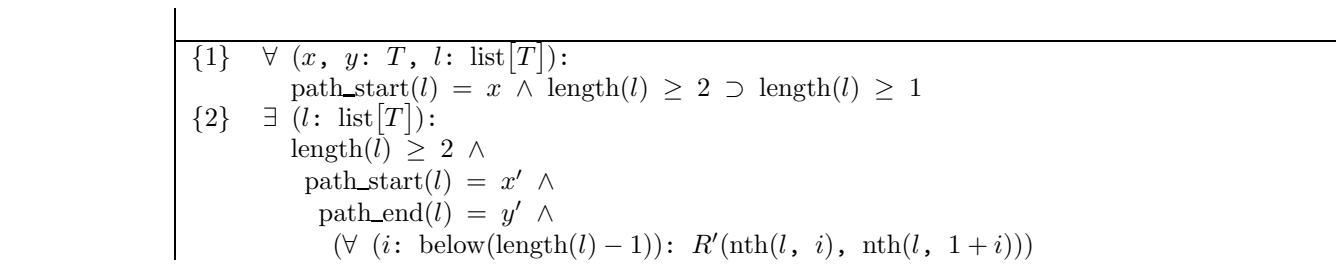
{-1}	$\text{length}(l') \geq 2$
{-2}	$\text{path\_start}(l') = x''$
{-3}	$\text{path\_end}(l') = y''$
{-4}	$\forall\ (i\colon\ \text{below}(\text{length}(l') - 1))\colon\ R'(\text{nth}(l',\ i),\ \text{nth}(l',\ 1 + i))$
{-5}	$\text{length}(l'') \geq 2$
{-6}	$\text{path\_start}(l'') = y''$
{-7}	$\text{path\_end}(l'') = z'$
{-8}	$\forall\ (i\colon\ \text{below}(\text{length}(l'') - 1))\colon\ R'(\text{nth}(l'',\ i),\ \text{nth}(l'',\ 1 + i))$
{1}	$\text{path\_end}(\text{append}(l',\ \text{cdr}(l''))) = z'$

Expanding the definition of path_end,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of nth,
which is trivially true.
This completes the proof of `trans_closure_char.1.1.2.2`.
`trans_closure_char.1.1.2.3:`

{-1}	$\text{length}(l') \geq 2$
{-2}	$\text{path\_start}(l') = x''$
{-3}	$\text{path\_end}(l') = y''$
{-4}	$\forall\ (i\colon\ \text{below}(\text{length}(l') - 1))\colon\ R'(\text{nth}(l',\ i),\ \text{nth}(l',\ 1 + i))$
{-5}	$\text{length}(l'') \geq 2$
{-6}	$\text{path\_start}(l'') = y''$
{-7}	$\text{path\_end}(l'') = z'$
{-8}	$\forall\ (i\colon\ \text{below}(\text{length}(l'') - 1))\colon\ R'(\text{nth}(l'',\ i),\ \text{nth}(l'',\ 1 + i))$
{1}	$\text{path\_start}(\text{append}(l',\ \text{cdr}(l''))) = x''$

Expanding the definition of path_start,
Expanding the definition of append,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `trans_closure_char.1.1.2.3`.

`trans_closure_char.1.2:`

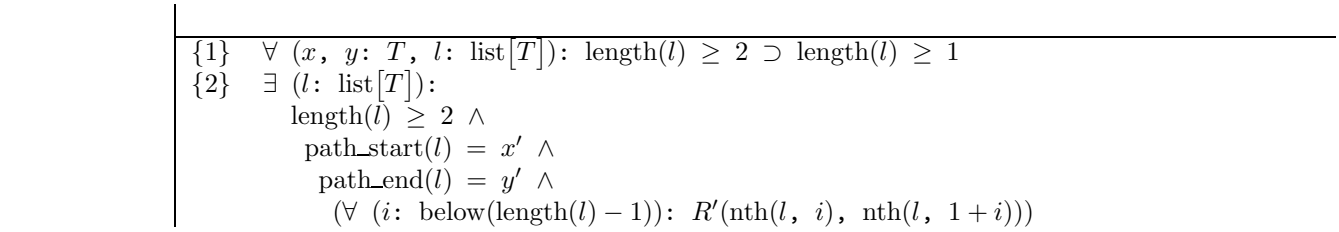


Hiding formulas: 2,

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `trans_closure_char.1.2`.

`trans_closure_char.1.3:`

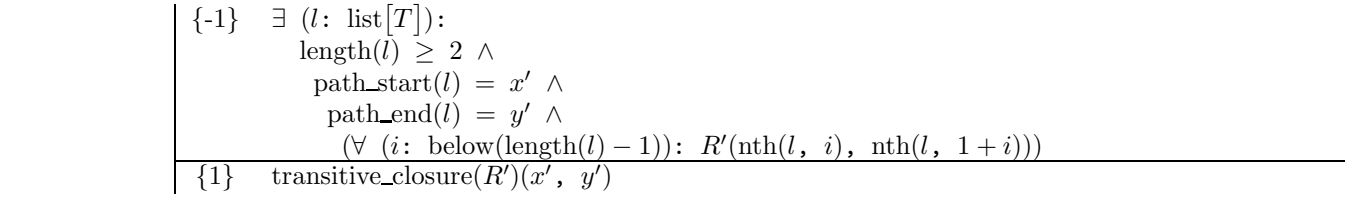


Hiding formulas: 2,

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `trans_closure_char.1.3`.

`trans_closure_char.2:`



Case splitting on FORALL (x, y: T, l: (path_list?[T])): length(l) >= 2 AND path_start(l) = x AND path_end(l) = y AND (FORALL (i: below(length(l) - 1)): R!1(nth(l, i), nth(l, 1 + i))) IMPLIES transitive_closure(R!1)(x, y),

we get 2 subgoals:

`trans_closure_char.2.1:`

{-1} $\forall\ (x,\ y\colon T,\ l\colon (\text{path\_list?}[T]))\colon$
$\text{length}(l) \geq 2\ \wedge$
$\text{path\_start}(l) = x\ \wedge$
$\text{path\_end}(l) = y\ \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(l)-1))\colon$
$R'(\text{nth}(l,\ i),\ \text{nth}(l,\ 1+i)))$
$\supset\ \text{transitive\_closure}(R')(x,\ y)$

{-2} $\exists\ (l\colon \text{list}[T])\colon$
$\text{length}(l) \geq 2\ \wedge$
$\text{path\_start}(l) = x'\ \wedge$
$\text{path\_end}(l) = y'\ \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(l)-1))\colon R'(\text{nth}(l,\ i),\ \text{nth}(l,\ 1+i)))$

{1} $\text{transitive\_closure}(R')(x',\ y')$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char.2.1`.

`trans_closure_char.2.2:`

{-1} $\exists\ (l\colon \text{list}[T])\colon$
$\text{length}(l) \geq 2\ \wedge$
$\text{path\_start}(l) = x'\ \wedge$
$\text{path\_end}(l) = y'\ \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(l)-1))\colon R'(\text{nth}(l,\ i),\ \text{nth}(l,\ 1+i)))$

{1} $\forall\ (x,\ y\colon T,\ l\colon (\text{path\_list?}[T]))\colon$
$\text{length}(l) \geq 2\ \wedge$
$\text{path\_start}(l) = x\ \wedge$
$\text{path\_end}(l) = y\ \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(l)-1))\colon$
$R'(\text{nth}(l,\ i),\ \text{nth}(l,\ 1+i)))$
$\supset\ \text{transitive\_closure}(R')(x,\ y)$

{2} $\text{transitive\_closure}(R')(x',\ y')$

Hiding formulas: -1, 2,
Inducting on l on formula 1 using induction scheme $\text{path\_list\_induction}[T]$,
we get 2 subgoals:

`trans_closure_char.2.2.1:`

{1} $\forall\ (t\colon T)\colon$
$\forall\ (x,\ y\colon T)\colon$
$(\text{length}(\text{cons}(t,\ \text{null})) \geq 2\ \wedge$
$\text{path\_start}(\text{cons}(t,\ \text{null})) = x\ \wedge$
$\text{path\_end}(\text{cons}(t,\ \text{null})) = y\ \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t,\ \text{null}))-1))\colon$
$R'(\text{nth}(\text{cons}(t,\ \text{null}),\ i),\ \text{nth}(\text{cons}(t,\ \text{null}),\ 1+i))))$
$\supset\ \text{transitive\_closure}(R')(x,\ y)$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: nth path_start path_end
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char.2.2.1`.

`trans_closure_char.2.2.2:`

```
{1}  ∀ (t: T, tail: (path_list?⌊T⌋)):
       (∀ (x, y: T):
          (length(tail) ≥ 2 ∧
             path_start(tail) = x ∧
              path_end(tail) = y ∧
               (∀ (i: below(length(tail) − 1)):
                    R′(nth(tail, i), nth(tail, 1 + i))))
           ⊃ transitive_closure(R′)(x, y))
        ⊃
        (∀ (x, y: T):
           (length(cons(t, tail)) ≥ 2 ∧
              path_start(cons(t, tail)) = x ∧
               path_end(cons(t, tail)) = y ∧
                (∀ (i: below(length(cons(t, tail)) − 1)):
                     R′(nth(cons(t, tail), i), nth(cons(t, tail), 1 + i))))
            ⊃ transitive_closure(R′)(x, y))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_start path_end
Simplifying, rewriting, and recording with decision procedures,
Case splitting on length(tail!1) = 1,
we get 2 subgoals:
`trans_closure_char.2.2.2.1:`

```
{-1}  length(tail′) = 1
{-2}  length(tail′) ≥ 1
{-3}  ∀ (x, y: T):
        (length(tail′) ≥ 2 ∧
           car(tail′) = x ∧
            nth(tail′, length(tail′) − 1) = y ∧
             (∀ (i: below(length(tail′) − 1)):
                  R′(nth(tail′, i), nth(tail′, 1 + i))))
         ⊃ transitive_closure(R′)(x, y)
{-4}  1 + length(tail′) ≥ 2
{-5}  t′ = x″
{-6}  nth(cons(t′, tail′), length(tail′)) = y″
{-7}  ∀ (i: below(length(cons(t′, tail′)) − 1)):
        R′(nth(cons(t′, tail′), i), nth(cons(t′, tail′), 1 + i))
------------------------------------------------------------------
{1}  transitive_closure(R′)(x″, y″)
```

Hiding formulas: -3,
Installing automatic rewrites from: nth
Instantiating the top quantifier in -6 with the terms: 0,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of transitive_closure,
Repeatedly Skolemizing and flattening,
Expanding the definition(s) of (subset? member),
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char.2.2.2.1`.

`trans_closure_char.2.2.2.2:`

{-1} $\text{length}(\text{tail}') \geq 1$
{-2} $\forall\ (x,\ y\colon T)\colon$
$(\text{length}(\text{tail}') \geq 2 \wedge$
$\text{car}(\text{tail}') = x \wedge$
$\text{nth}(\text{tail}',\ \text{length}(\text{tail}') - 1) = y \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(\text{tail}') - 1))\colon$
$R'(\text{nth}(\text{tail}',\ i),\ \text{nth}(\text{tail}',\ 1 + i))))$
$\supset\ \text{transitive\_closure}(R')(x,\ y)$
{-3} $1 + \text{length}(\text{tail}') \geq 2$
{-4} $t' = x''$
{-5} $\text{nth}(\text{cons}(t',\ \text{tail}'),\ \text{length}(\text{tail}')) = y''$
{-6} $\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t',\ \text{tail}')) - 1))\colon$
$R'(\text{nth}(\text{cons}(t',\ \text{tail}'),\ i),\ \text{nth}(\text{cons}(t',\ \text{tail}'),\ 1 + i))$

{1} $\text{length}(\text{tail}') = 1$
{2} $\text{transitive\_closure}(R')(x'',\ y'')$

Expanding the definition of nth,
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in -2 with the terms: $\text{car}(\text{tail}')$, y'',
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`trans_closure_char.2.2.2.2.1:`

{-1} $\text{length}(\text{tail}') \geq 1$
{-2} $\text{nth}(\text{tail}',\ \text{length}(\text{tail}') - 1) = y''$
{-3} $\text{transitive\_closure}(R')(\text{car}(\text{tail}'),\ y'')$
{-4} $1 + \text{length}(\text{tail}') \geq 2$
{-5} $t' = x''$
{-6} $\forall\ (i\colon \text{below}(\text{length}(\text{cons}(t',\ \text{tail}')) - 1))\colon$
$R'(\text{nth}(\text{cons}(t',\ \text{tail}'),\ i),\ \text{nth}(\text{cons}(t',\ \text{tail}'),\ 1 + i))$

{1} $\text{length}(\text{tail}') = 1$
{2} $\text{transitive\_closure}(R')(x'',\ y'')$

Instantiating the top quantifier in -6 with the terms: 0,
Installing automatic rewrites from: nth
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -5,
Hiding formulas: -1, -2, -4, -5,
Expanding the definition of transitive_closure,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of transitive?,
Instantiating the top quantifier in -4 with the terms: x'', $\text{car}(\text{tail}')$, y'',
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition(s) of (subset? member),
Instantiating the top quantifier in -3 with the terms: $(x'', \text{car}(\text{tail}'))$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char.2.2.2.2.1`.

`trans_closure_char.2.2.2.2.2:`

{-1}	length(tail′) ≥ 1
{-2}	nth(tail′, length(tail′) − 1) = y''
{-3}	1 + length(tail′) ≥ 2
{-4}	t' = x''
{-5}	∀ (i: below(length(cons(t', tail′)) − 1)): R'(nth(cons(t', tail′), i), nth(cons(t', tail′), 1 + i))
{1}	length(tail′) = 1
{2}	∀ (i: below(length(tail′) − 1)): R'(nth(tail′, i), nth(tail′, 1 + i))
{3}	transitive_closure(R')(x'', y'')

Keeping (-5 2) and hiding *,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: $i' + 1$,
Installing automatic rewrites from: nth
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `trans_closure_char.2.2.2.2.2`.
Q.E.D.

C.99 Proofs for Graph_Util (graph.pvs)

C.99.1 Graph_Util.every_nth

Terse proof for `every_nth`.
`every_nth:`

{1}	∀ (P: PRED[T], l: list[T]): every(P)(l) ≡ (∀ (i: below(length(l))): P(nth(l, i)))

Inducting on l on formula 1,
we get 2 subgoals:
`every_nth.1:`

{1}	∀ (P: PRED[T]): every(P)(null) ≡ (∀ (i: below(length(null))): P(nth(null, i)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `every_nth.1`.
`every_nth.2:`

{1}	∀ (cons1_var: T, cons2_var: list[T]): (∀ (P: PRED[T]): every(P)(cons2_var) ≡ (∀ (i: below(length(cons2_var))): P(nth(cons2_var, i)))) ⊃ (∀ (P: PRED[T]): every(P)(cons(cons1_var, cons2_var)) ≡ (∀ (i: below(length(cons(cons1_var, cons2_var)))): P(nth(cons(cons1_var, cons2_var), i))))

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:
`every_nth.2.1:`

```
{-1}  every(P')(cons(cons1_var', cons2_var'))
{-2}  ∀ (P: PRED[T]):
        every(P)(cons2_var') ≡
          (∀ (i: below(length(cons2_var'))): P(nth(cons2_var', i)))
------------------------------------------------------------------
{1}   ∀ (i: below(length(cons(cons1_var', cons2_var')))):
        P'(nth(cons(cons1_var', cons2_var'), i))
```

Expanding the definition of every,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Expanding the definition of length,
Expanding the definition of nth,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -4 with the terms: $i' - 1$,
This completes the proof of `every_nth.2.1`.
`every_nth.2.2:`

```
{-1}  ∀ (i: below(length(cons(cons1_var', cons2_var')))):
        P'(nth(cons(cons1_var', cons2_var'), i))
{-2}  ∀ (P: PRED[T]):
        every(P)(cons2_var') ≡
          (∀ (i: below(length(cons2_var'))): P(nth(cons2_var', i)))
------------------------------------------------------------------
{1}   every(P')(cons(cons1_var', cons2_var'))
```

Expanding the definition of every,
Applying propositional simplification,
we get 2 subgoals:
`every_nth.2.2.1:`

```
{-1}  ∀ (i: below(length(cons(cons1_var', cons2_var')))):
        P'(nth(cons(cons1_var', cons2_var'), i))
{-2}  ∀ (P: PRED[T]):
        every(P)(cons2_var') ≡
          (∀ (i: below(length(cons2_var'))): P(nth(cons2_var', i)))
------------------------------------------------------------------
{1}   P'(cons1_var')
```

Instantiating the top quantifier in -1 with the terms: 0,
we get 2 subgoals:
`every_nth.2.2.1.1:`

```
{-1}  P'(nth(cons(cons1_var', cons2_var'), 0))
{-2}  ∀ (P: PRED[T]):
        every(P)(cons2_var') ≡
          (∀ (i: below(length(cons2_var'))): P(nth(cons2_var', i)))
------------------------------------------------------------------
{1}   P'(cons1_var')
```

Expanding the definition of nth,
which is trivially true.
This completes the proof of `every_nth.2.2.1.1`.

every_nth.2.2.1.2:

{-1}	$\forall$ (P: PRED$[T]$): every(P)(cons2_var′) $\equiv$ ($\forall$ (i: below(length(cons2_var′))): P(nth(cons2_var′, i)))
{1}	0 $<$ length$[T]$(cons$[T]$(cons1_var′, cons2_var′))
{2}	P'(cons1_var′)

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of every_nth.2.2.1.2.
every_nth.2.2.2:

{-1}	$\forall$ (i: below(length(cons(cons1_var′, cons2_var′)))): P'(nth(cons(cons1_var′, cons2_var′), i))
{-2}	$\forall$ (P: PRED$[T]$): every(P)(cons2_var′) $\equiv$ ($\forall$ (i: below(length(cons2_var′))): P(nth(cons2_var′, i)))
{1}	every(P')(cons2_var′)

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: 1,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: $i' + 1$,
we get 2 subgoals:
every_nth.2.2.2.1:

{-1}	i' $<$ length(cons2_var′)
{-2}	P'(nth(cons(cons1_var′, cons2_var′), $i' + 1$))
{1}	P'(nth(cons2_var′, i'))

Expanding the definition of nth,
which is trivially true.
This completes the proof of every_nth.2.2.2.1.
every_nth.2.2.2.2:

{-1}	i' $<$ length(cons2_var′)
{1}	$1 + i'$ $<$ length$[T]$(cons$[T]$(cons1_var′, cons2_var′))
{2}	P'(nth(cons2_var′, i'))

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of every_nth.2.2.2.2.
Q.E.D.

C.99.2 Graph_Util.nth_append_left_TCC1

Terse proof for nth_append_left_TCC1.
nth_append_left_TCC1:

{1}	$\forall$ (l_1, l_2: list$[T]$, n: nat): n $<$ length(l_1) $\supset$ n $<$ length$[T]$(append$[T]$(l_1, l_2))

Repeatedly Skolemizing and flattening,
Rewriting using length_append, matching in *,

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_append_left_TCC1`.
Q.E.D.

C.99.3 Graph_Util.nth_append_left

Terse proof for `nth_append_left`.
`nth_append_left:`

{1} ∀ (l_1, l_2: list[T], n: nat):
n < length(l_1) ⊃ nth(append(l_1, l_2), n) = nth(l_1, n)

Inducting on l_1 on formula 1,
we get 3 subgoals:
`nth_append_left.1:`

{1} ∀ (l_2: list[T], n: nat):
n < length(null) ⊃ nth(append(null, l_2), n) = nth(null, n)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_append_left.1`.
`nth_append_left.2:`

{1} ∀ (cons1_var: T, cons2_var: list[T]):
(∀ (l_2: list[T], n: nat):
n < length(cons2_var) ⊃ nth(append(cons2_var, l_2), n) = nth(cons2_var, n))
⊃
(∀ (l_2: list[T], n: nat):
n < length(cons(cons1_var, cons2_var)) ⊃
nth(append(cons(cons1_var, cons2_var), l_2), n) =
nth(cons(cons1_var, cons2_var), n))

Repeatedly Skolemizing and flattening,
Expanding the definition of append,
Expanding the definition of nth,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using -2, matching in *,
This completes the proof of `nth_append_left.2`.
`nth_append_left.3:`

{1} ∀ (l_1, l_2: list[T], n: nat):
n < length(l_1) ⊃ n < length[T](append[T](l_1, l_2))
{2} ∀ (l_2: list[T], n: nat):
n < length(l'_1) ⊃ nth(append(l'_1, l_2), n) = nth(l'_1, n)

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using length_append, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_append_left.3`.
Q.E.D.

C.99.4 Graph_Util.nth_append_right_TCC1

Terse proof for `nth_append_right_TCC1`.

`nth_append_right_TCC1:`

{1} $\forall\ (l_1,\ l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(l_1) \wedge n < \text{length}(\text{append}(l_1,\ l_2)) \supset$
$n - \text{length}[T](l_1) \geq 0 \wedge n - \text{length}[T](l_1) < \text{length}[T](l_2)$

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using length_append, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_append_right_TCC1`.
Q.E.D.

C.99.5 Graph_Util.nth_append_right

Terse proof for `nth_append_right`.

`nth_append_right:`

{1} $\forall\ (l_1,\ l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(l_1) \wedge n < \text{length}(\text{append}(l_1,\ l_2)) \supset$
$\text{nth}(\text{append}(l_1,\ l_2),\ n) = \text{nth}(l_2,\ n - \text{length}(l_1))$

Inducting on l_1 on formula 1,
we get 3 subgoals:

`nth_append_right.1:`

{1} $\forall\ (l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(\text{null}) \wedge n < \text{length}(\text{append}(\text{null},\ l_2)) \supset$
$\text{nth}(\text{append}(\text{null},\ l_2),\ n) = \text{nth}(l_2,\ n - \text{length}(\text{null}))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_append_right.1`.

`nth_append_right.2:`

{1} $\forall\ (\text{cons1\_var}\colon T,\ \text{cons2\_var}\colon \text{list}[T])\colon$
$(\forall\ (l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(\text{cons2\_var}) \wedge n < \text{length}(\text{append}(\text{cons2\_var},\ l_2)) \supset$
$\text{nth}(\text{append}(\text{cons2\_var},\ l_2),\ n) = \text{nth}(l_2,\ n - \text{length}(\text{cons2\_var})))$
$\supset$
$(\forall\ (l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(\text{cons}(\text{cons1\_var},\ \text{cons2\_var})) \wedge$
$n < \text{length}(\text{append}(\text{cons}(\text{cons1\_var},\ \text{cons2\_var}),\ l_2))$
$\supset$
$\text{nth}(\text{append}(\text{cons}(\text{cons1\_var},\ \text{cons2\_var}),\ l_2),\ n) =$
$\text{nth}(l_2,\ n - \text{length}(\text{cons}(\text{cons1\_var},\ \text{cons2\_var}))))$

Repeatedly Skolemizing and flattening,
Rewriting using length_append, matching in *,
Expanding the definition of length,

Expanding the definition of length,
Expanding the definition of append,
Expanding the definition of nth,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using -2, matching in *,
Rewriting using length_append, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_append_right.2`.
`nth_append_right.3`:

{1} $\forall\ (l_1,\ l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(l_1) \wedge n < \text{length}(\text{append}(l_1,\ l_2)) \supset$
$n - \text{length}[T](l_1) \geq 0 \wedge n - \text{length}[T](l_1) < \text{length}[T](l_2)$
{2} $\forall\ (l_2\colon \text{list}[T],\ n\colon \text{nat})\colon$
$n \geq \text{length}(l_1') \wedge n < \text{length}(\text{append}(l_1',\ l_2)) \supset$
$\text{nth}(\text{append}(l_1',\ l_2),\ n) = \text{nth}(l_2,\ n - \text{length}(l_1'))$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using length_append, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_append_right.3`.
Q.E.D.

C.99.6 Graph_Util.list_pred_card

Terse proof for `list_pred_card`.
`list_pred_card`:

{1} $\forall\ (l\colon \text{list}[T],\ P\colon \text{PRED}[T])\colon$
$\text{is\_finite}(P) \wedge \text{every}(P)(l) \wedge \text{length}(l) > \text{card}(P) \supset$
$(\exists\ (n_1,\ n_2\colon \text{below}(\text{length}(l)))\colon \neg\ n_1 = n_2 \wedge \text{nth}(l,\ n_1) = \text{nth}(l,\ n_2))$

Repeatedly Skolemizing and flattening,
Letting f name LAMBDA (i: below(length(l!1))): nth(l!1, i),
Case splitting on injective?(f),
we get 2 subgoals:
`list_pred_card.1`:

{-1} injective?(f)
{-2} $(\lambda\ (i\colon \text{below}(\text{length}(l'))\colon \text{nth}(l',\ i)) = f$
{-3} is_finite(P')
{-4} every$(P')(l')$
{-5} length$(l') >$ card(P')
{1} $\exists\ (n_1,\ n_2\colon \text{below}(\text{length}(l')))\colon \neg\ n_1 = n_2 \wedge \text{nth}(l',\ n_1) = \text{nth}(l',\ n_2)$

Using lemma bijective_restrict_to_image,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on (EXISTS (d: below(length[T](l!1))): TRUE) OR (FORALL (r: (image[below(length(l!1)), T] (f, fullset[below(length[T](l!1))]))): FALSE),
we get 2 subgoals:

`list_pred_card.1.1:`

{-1} $(\exists\ (d\colon\ \text{below}(\text{length}[T](l'))):\ \text{TRUE})\ \vee$
$(\forall\ (r\colon$
$(\text{image}[\text{below}(\text{length}(l')),\ T](f,\ \text{fullset}[\text{below}(\text{length}[T](l'))]))):$
FALSE)
{-2} bijective?(restrict_to_image(f))
{-3} injective?(f)
{-4} $(\lambda\ (i\colon\ \text{below}(\text{length}(l'))):\ \text{nth}(l',\ i)) = f$
{-5} is_finite(P')
{-6} every(P')(l')
{-7} $\text{length}(l') > \text{card}(P')$

{1} $\exists\ (n_1,\ n_2\colon\ \text{below}(\text{length}(l'))):\ \neg\ n_1 = n_2\ \wedge\ \text{nth}(l',\ n_1) = \text{nth}(l',\ n_2)$

Using lemma bijective_inverse_alt_is_bijective,

Using lemma card_bij,

we get 2 subgoals:

`list_pred_card.1.1.1:`

{-1} $\text{card}(\text{image}(f,\ \text{fullset}[\text{below}(\text{length}(l'))])) = \text{length}(l') \equiv$
$(\exists\ (\text{f\_1}\colon\ [(\text{image}(f,\ \text{fullset}[\text{below}(\text{length}(l'))])) \rightarrow \text{below}[\text{length}(l')]]):$
bijective?(f_1))
{-2} bijective?$[(\text{image}(f,\ \text{fullset}[\text{below}(\text{length}(l'))])),\ \text{below}(\text{length}(l'))]$
(inverse_alt(restrict_to_image(f)))
{-3} $(\exists\ (d\colon\ \text{below}(\text{length}[T](l'))):\ \text{TRUE})\ \vee$
$(\forall\ (r\colon$
$(\text{image}[\text{below}(\text{length}(l')),\ T](f,\ \text{fullset}[\text{below}(\text{length}[T](l'))]))):$
FALSE)
{-4} bijective?(restrict_to_image(f))
{-5} injective?(f)
{-6} $(\lambda\ (i\colon\ \text{below}(\text{length}(l'))):\ \text{nth}(l',\ i)) = f$
{-7} is_finite(P')
{-8} every(P')(l')
{-9} $\text{length}(l') > \text{card}(P')$

{1} $\exists\ (n_1,\ n_2\colon\ \text{below}(\text{length}(l'))):\ \neg\ n_1 = n_2\ \wedge\ \text{nth}(l',\ n_1) = \text{nth}(l',\ n_2)$

Applying disjunctive simplification to flatten sequent,

Splitting conjunctions,

we get 2 subgoals:

`list_pred_card.1.1.1.1:`

```
{-1}  card(image(f, fullset[below(length(l'))])) = length(l')
{-2}  card(image(f, fullset[below(length(l'))])) = length(l') ⊃
        (∃ (f_1: [(image(f, fullset[below(length(l'))])) → below[length(l')]]):
           bijective?(f_1))
{-3}  bijective?[(image(f, fullset[below(length(l'))])), below(length(l'))]
        (inverse_alt(restrict_to_image(f)))
{-4}  (∃ (d: below(length[T](l'))): TRUE) ∨
        (∀ (r:
              (image[below(length(l')), T](f, fullset[below(length[T](l'))]))):
           FALSE)
{-5}  bijective?(restrict_to_image(f))
{-6}  injective?(f)
{-7}  (λ (i: below(length(l'))): nth(l', i)) = f
{-8}  is_finite(P')
{-9}  every(P')(l')
{-10} length(l') > card(P')
------------------------------------------------------------------------
 {1}  ∃ (n_1, n_2: below(length(l'))): ¬ n_1 = n_2 ∧ nth(l', n_1) = nth(l', n_2)
```

Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -2,
Using lemma card_subset,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -1, -2, -3, -4, -5, -6, 2,
Expanding the definition(s) of (fullset subset? member image f),
Repeatedly Skolemizing and flattening,
Rewriting using every_nth, matching in *,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `list_pred_card.1.1.1.1`.
`list_pred_card.1.1.1.2:`

```
{-1}  card(image(f, fullset[below(length(l'))])) = length(l') ⊃
        (∃ (f_1: [(image(f, fullset[below(length(l'))])) → below[length(l')]]):
           bijective?(f_1))
{-2}  bijective?[(image(f, fullset[below(length(l'))])), below(length(l'))]
        (inverse_alt(restrict_to_image(f)))
{-3}  (∃ (d: below(length[T](l'))): TRUE) ∨
        (∀ (r:
              (image[below(length(l')), T](f, fullset[below(length[T](l'))]))):
           FALSE)
{-4}  bijective?(restrict_to_image(f))
{-5}  injective?(f)
{-6}  (λ (i: below(length(l'))): nth(l', i)) = f
{-7}  is_finite(P')
{-8}  every(P')(l')
{-9}  length(l') > card(P')
------------------------------------------------------------------------
 {1}  ∃ (f_1: [(image(f, fullset[below(length(l'))])) → below[length(l')]]):
        bijective?(f_1)
 {2}  ∃ (n_1, n_2: below(length(l'))): ¬ n_1 = n_2 ∧ nth(l', n_1) = nth(l', n_2)
```

Hiding formulas: -1, 2,
Instantiating quantified variables,
This completes the proof of `list_pred_card.1.1.1.2`.

`list_pred_card.1.1.2:`

{-1} bijective?[(image(f, fullset[below(length(l'))])), below(length(l'))]
(inverse_alt(restrict_to_image(f)))
{-2} ($\exists$ (d: below(length[T](l'))): TRUE) $\vee$
($\forall$ (r:
(image[below(length(l')), T](f, fullset[below(length[T](l'))]))):
FALSE)
{-3} bijective?(restrict_to_image(f))
{-4} injective?(f)
{-5} (λ (i: below(length(l'))): nth(l', i)) = f
{-6} is_finite(P')
{-7} every(P')(l')
{-8} length(l') > card(P')

{1} is_finite[T]
(image[below(length(l')), T](f, fullset[below(length[T](l'))]))
{2} $\exists$ (n_1, n_2: below(length(l'))): $\neg$ n_1 = n_2 $\wedge$ nth(l', n_1) = nth(l', n_2)

Hiding formulas: -1, -2, 2,
Using lemma finite_image,
Keeping 1 and hiding *,
Expanding the definition of is_finite,
Instantiating the top quantifier in 1 with the terms: length(l'), id,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `list_pred_card.1.1.2`.
`list_pred_card.1.2:`

{-1} bijective?(restrict_to_image(f))
{-2} injective?(f)
{-3} (λ (i: below(length(l'))): nth(l', i)) = f
{-4} is_finite(P')
{-5} every(P')(l')
{-6} length(l') > card(P')

{1} ($\exists$ (d: below(length[T](l'))): TRUE) $\vee$
($\forall$ (r:
(image[below(length(l')), T](f, fullset[below(length[T](l'))]))):
FALSE)
{2} $\exists$ (n_1, n_2: below(length(l'))): $\neg$ n_1 = n_2 $\wedge$ nth(l', n_1) = nth(l', n_2)

Applying disjunctive simplification to flatten sequent,
Hiding formulas: 3,
Repeatedly Skolemizing and flattening,
Expanding the definition of image,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `list_pred_card.1.2`.
`list_pred_card.2:`

{-1} (λ (i: below(length(l'))): nth(l', i)) = f
{-2} is_finite(P')
{-3} every(P')(l')
{-4} length(l') > card(P')

{1} injective?(f)
{2} $\exists$ (n_1, n_2: below(length(l'))): $\neg$ n_1 = n_2 $\wedge$ nth(l', n_1) = nth(l', n_2)

Expanding the definition of injective?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 2 with the terms: x'_1, x'_2,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -4,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `list_pred_card.2`.
Q.E.D.

C.99.7 Graph_Util.path_start_TCC1

Terse proof for `path_start_TCC1`.
`path_start_TCC1`:

$$\vdash$$
$$\{1\} \quad \forall\ (l\colon\ (\text{path\_list?}))\colon\ \text{cons?}[T](l)$$

Repeatedly Skolemizing and flattening,
Expanding the definition of path_list?,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `path_start_TCC1`.
Q.E.D.

C.99.8 Graph_Util.path_end_TCC1

Terse proof for `path_end_TCC1`.
`path_end_TCC1`:

$$\vdash$$
$$\{1\} \quad \forall\ (l\colon\ (\text{path\_list?}))\colon \\ \quad \text{length}[T](l) - 1\ \geq\ 0\ \wedge\ \text{length}[T](l) - 1\ <\ \text{length}[T](l)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `path_end_TCC1`.
Q.E.D.

C.99.9 Graph_Util.path_list_induction_TCC1

Terse proof for `path_list_induction_TCC1`.
`path_list_induction_TCC1`:

$$\vdash$$
$$\{1\} \quad \forall\ (t\colon\ T)\colon\ \text{path\_list?}(\text{cons}[T](t,\ \text{null}[T]))$$

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_list? length
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_list_induction_TCC1`.
Q.E.D.

C.99.10 Graph_Util.path_list_induction_TCC2

Terse proof for `path_list_induction_TCC2`.

`path_list_induction_TCC2:`

{1}	∀ (p: [(path_list?) → boolean]): (∀ (t: T): p(cons(t, null))) ⊃ (∀ (t_1: T, tail: (path_list?)): p(tail) ⊃ path_list?(cons[T](t_1, tail)))

Repeatedly Skolemizing and flattening,
Hiding formulas: -2, -3,
Installing automatic rewrites from: path_list? length
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_list_induction_TCC2`.
Q.E.D.

C.99.11 Graph_Util.path_list_induction

Terse proof for `path_list_induction`.

`path_list_induction:`

{1}	∀ (p: [(path_list?) → boolean]): (∀ (t: T): p(cons(t, null))) ∧ (∀ (t: T, tail: (path_list?)): p(tail) ⊃ p(cons(t, tail))) ⊃ (∀ (l: (path_list?)): p(l))

Skolemizing and flattening,
Inducting on l on formula 1,
we get 3 subgoals:

`path_list_induction.1:`

{-1}	∀ (t: T): p'(cons(t, null))
{-2}	∀ (t: T, tail: (path_list?)): p'(tail) ⊃ p'(cons(t, tail))
{1}	path_list?(l')
{2}	p'(l')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_list_induction.1`.

`path_list_induction.2:`

{-1}	∀ (t: T): p'(cons(t, null))
{-2}	∀ (t: T, tail: (path_list?)): p'(tail) ⊃ p'(cons(t, tail))
{1}	path_list?(null) ⊃ p'(null)

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_list? length
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_list_induction.2`.

`path_list_induction.3:`

{-1}	∀ (t: T): p'(cons(t, null))
{-2}	∀ (t: T, tail: (path_list?)): p'(tail) ⊃ p'(cons(t, tail))
{1}	∀ (cons1_var: T, cons2_var: list[T]): (path_list?(cons2_var) ⊃ p'(cons2_var)) ⊃ path_list?(cons(cons1_var, cons2_var)) ⊃ p'(cons(cons1_var, cons2_var))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: path_list? length
Simplifying, rewriting, and recording with decision procedures,
Case splitting on cons2_var!1 = null,
we get 2 subgoals:
`path_list_induction.3.1:`

{-1} cons2_var′ = null
{-2} length(cons2_var′) ≥ 1 ⊃ p′(cons2_var′)
{-3} 1 + length(cons2_var′) ≥ 1
{-4} ∀ (t: T): p′(cons(t, null))
{-5} ∀ (t: T, tail: (path_list?)): p′(tail) ⊃ p′(cons(t, tail))

{1} p′(cons(cons1_var′, cons2_var′))

Hiding formulas: -2, -3,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `path_list_induction.3.1`.
`path_list_induction.3.2:`

{-1} length(cons2_var′) ≥ 1 ⊃ p′(cons2_var′)
{-2} 1 + length(cons2_var′) ≥ 1
{-3} ∀ (t: T): p′(cons(t, null))
{-4} ∀ (t: T, tail: (path_list?)): p′(tail) ⊃ p′(cons(t, tail))

{1} cons2_var′ = null
{2} p′(cons(cons1_var′, cons2_var′))

Simplifying, rewriting, and recording with decision procedures,
Rewriting using -4, matching in *,
This completes the proof of `path_list_induction.3.2`.
Q.E.D.

C.99.12 Graph_Util.concat_path_TCC1

Terse proof for `concat_path_TCC1`.
`concat_path_TCC1:`

{1} ∀ (lr: (concatable?)): cons?[T](PROJ_2(lr))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `concat_path_TCC1`.
Q.E.D.

C.99.13 Graph_Util.concat_path_TCC2

Terse proof for `concat_path_TCC2`.
`concat_path_TCC2:`

{1} ∀ (lr: (concatable?)): path_list?(append[T](PROJ_1(lr), cdr[T](PROJ_2(lr))))

Repeatedly Skolemizing and flattening,
Expanding the definition of concatable?,
Applying disjunctive simplification to flatten sequent,

Hiding formulas: -3,
Expanding the definition of path_list?,
Rewriting using length_append, matching in *,
Rewriting using length_cdr, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `concat_path_TCC2`.
Q.E.D.

C.99.14 Graph_Util.length_concat_path

Terse proof for `length_concat_path`.
`length_concat_path:`

{1}	$\forall$ (l, r: list$[T]$): concatable?(l, r) $\supset$ length(concat_path(l, r)) = length(l) + length(r) $-$ 1

Repeatedly Skolemizing and flattening,
Expanding the definition of concat_path,
Rewriting using length_append, matching in *,
Rewriting using length_cdr, matching in *,
we get 2 subgoals:
`length_concat_path.1:`

{-1}	concatable?(l', r')
{1}	length(r') $-$ 1 + length(l') = length(l') + length(r') $-$ 1

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `length_concat_path.1`.
`length_concat_path.2:`

{-1}	concatable?(l', r')
{1}	length(r') > 0
{2}	length(cdr(r')) = length(r') $-$ 1

Expanding the definition of concatable?,
Expanding the definition of path_list?,
which is trivially true.
This completes the proof of `length_concat_path.2`.
Q.E.D.

C.100 Proofs for Hoare (hoare.pvs)

C.100.1 Hoare.valid_TCC1

Terse proof for `valid_TCC1`.

`valid_TCC1:`

{1} ∀ (P: PRED[State], c: [State → StmtResult[State]], s):
$P(s)$ ∧ OK?($c(s)$) ⊃
OK?[State]($c(s)$) ∨
Break?[State]($c(s)$) ∨
Continue?[State]($c(s)$) ∨
Return?[State]($c(s)$) ∨
Switch?[State]($c(s)$) ∨
Default?[State]($c(s)$) ∨ Exception?[State]($c(s)$)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_TCC1`.
Q.E.D.

C.100.2 Hoare.valid_TCC2

Terse proof for `valid_TCC2`.

`valid_TCC2:`

{1} ∀ (P: PRED[State], C: [StmtResult[State] → StmtResult[State]], s):
$P(s)$ ∧ OK?(C(OK(s))) ⊃
OK?[State](C(OK[State](s))) ∨
Break?[State](C(OK[State](s))) ∨
Continue?[State](C(OK[State](s))) ∨
Return?[State](C(OK[State](s))) ∨
Switch?[State](C(OK[State](s))) ∨
Default?[State](C(OK[State](s))) ∨
Exception?[State](C(OK[State](s)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_TCC2`.
Q.E.D.

C.100.3 Hoare.valid_and

Terse proof for `valid_and`.

`valid_and:`

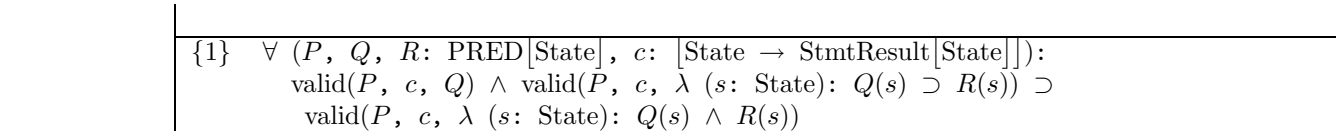
{1} ∀ (P, Q, R: PRED[State], c: [State → StmtResult[State]]):
valid(P, c, Q) ∧ valid(P, c, R) ⊃
valid(P, c, λ (s: State): $Q(s)$ ∧ $R(s)$)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_and`.
Q.E.D.

C.100.4 Hoare.valid_implied

Terse proof for `valid_implied`.

`valid_implied`:

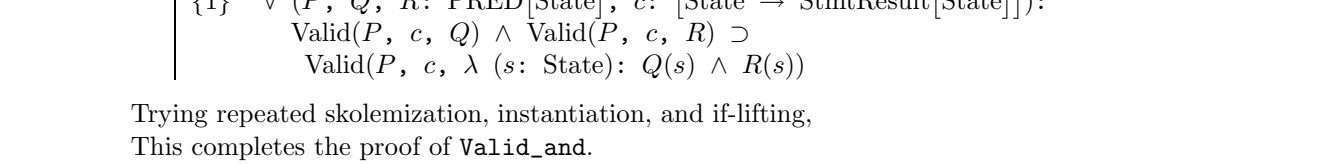


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_implied`.
Q.E.D.

C.100.5 Hoare.Valid_and

Terse proof for `Valid_and`.

`Valid_and`:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Valid_and`.
Q.E.D.

C.101 Proofs for IA32 (constants.pvs)

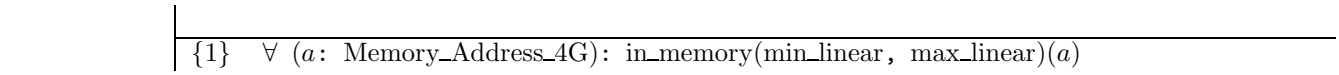
C.101.1 IA32.max_linear_offset_val

The LaTeX code for this proof is broken.

C.101.2 IA32.mem_addr_4g_TCC1

Terse proof for `mem_addr_4g_TCC1`.

`mem_addr_4g_TCC1`:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `mem_addr_4g_TCC1`.
Q.E.D.

C.102 Proofs for Import_All (everything.pvs)

This theory contains no provable formal statements.

C.103 Proofs for IntegralConversions (conversions.pvs)

This theory contains no provable formal statements.

C.104 Proofs for IntegralPromotions (conversions.pvs)

C.104.1 IntegralPromotions.int_conversion_TCC1

Terse proof for `int_conversion_TCC1`.

`int_conversion_TCC1:`

```
{1}  ∃ (x:
         [typ:
            Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                                 ({t: Cpp_Type |
                                     char?(t) ∨
                                       schar?(t) ∨
                                         uchar?(t) ∨ short?(t) ∨ ushort?(t)})) →
                          [(range(typ)) → (range(int))]]):
       TRUE
```

Instantiating the top quantifier in 1 with the terms: (λ (typ: Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] ({t: Cpp_Type | char?(t) ∨ schar?(t) ∨ uchar?(t) ∨ short?(t) ∨ ushort?(t)})))): λ (r: (range(typ))): 0),

we get 2 subgoals:

`int_conversion_TCC1.1:`

```
{1}  TRUE
```

which is trivially true.

This completes the proof of `int_conversion_TCC1.1`.

`int_conversion_TCC1.2:`

```
{1}  ∀ (typ:
          Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                               ({t: Cpp_Type |
                                   char?(t) ∨
                                     schar?(t) ∨
                                       uchar?(t) ∨ short?(t) ∨ ushort?(t)})),
             r: (range(typ))):
        range(int)(0)
```

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Rewriting using int_binary, matching in *,

Using lemma min_int,

Using lemma max_int,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `int_conversion_TCC1.2`.

Q.E.D.

C.104.2 IntegralPromotions.int_conversion_spec_TCC1

Terse proof for `int_conversion_spec_TCC1`.

`int_conversion_spec_TCC1:`

```
{1}  ∀ (typ:
           Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                                (({t: Cpp_Type |
                                       char?(t) ∨
                                        schar?(t) ∨
                                          uchar?(t) ∨ short?(t) ∨ ushort?(t)})),
                          n: (range(typ))):
       Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)
```

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:
`int_conversion_spec_TCC1.1:`

```
{1}  Cpp_Type?(int)
```

Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `int_conversion_spec_TCC1.1`.
`int_conversion_spec_TCC1.2:`

```
{1}  cv(non_bool_integral_enum?)(int)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `int_conversion_spec_TCC1.2`.
Q.E.D.

C.104.3 IntegralPromotions.int_conversion_TCC2

Terse proof for `int_conversion_TCC2`.
`int_conversion_TCC2:`

```
{1}  ∀ (typ:
           Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                                (({t: Cpp_Type |
                                       char?(t) ∨
                                        schar?(t) ∨
                                          uchar?(t) ∨ short?(t) ∨ ushort?(t)}))):
       cv(non_bool_integral_enum?)(typ)
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `int_conversion_TCC2`.
Q.E.D.

C.104.4 IntegralPromotions.int_conversion_TCC3

Terse proof for `int_conversion_TCC3`.

`int_conversion_TCC3:`

{1} ∀ (typ:
Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
({t: Cpp_Type |
char?(t) ∨
schar?(t) ∨
uchar?(t) ∨ short?(t) ∨ ushort?(t)}))):
Cpp_Type?(int) ∧ cv(non_bool_integral_enum?)(int)

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Applying propositional simplification,
we get 2 subgoals:

`int_conversion_TCC3.1:`

{1} Cpp_Type?(int)

Expanding the definition of Cpp_Type?,
Expanding the definition of subterm,
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `int_conversion_TCC3.1`.

`int_conversion_TCC3.2:`

{1} cv(non_bool_integral_enum?)(int)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `int_conversion_TCC3.2`.
Q.E.D.

C.104.5 IntegralPromotions.bool2int_TCC1

Terse proof for `bool2int_TCC1`.

`bool2int_TCC1:`

{1} ∀ (b: bool): b ⊃ range(int)(1)

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Expanding the definition of range_integral,
Rewriting using int_binary, matching in *,
Using lemma min_int,
Using lemma max_int,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bool2int_TCC1`.
Q.E.D.

C.104.6 IntegralPromotions.bool2int_TCC2

Terse proof for `bool2int_TCC2`.

`bool2int_TCC2:`

{1} ∀ (b: bool): ¬ b ⊃ range(int)(0)

Repeatedly Skolemizing and flattening,
Expanding the definition of range,
Expanding the definition of range_integral,
Rewriting using int_binary, matching in *,
Using lemma max_int,
Using lemma min_int,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `bool2int_TCC2`.
Q.E.D.

C.105 Proofs for Interpreted_Data (abstract_data.pvs)

C.105.1 Interpreted_Data.interpreted_data_type?_TCC1

Terse proof for `interpreted_data_type?_TCC1`.

`interpreted_data_type?_TCC1:`

{1} ∀ (idt: Interpreted_data_type):
(∀ (l: list[Byte], a: Address):
valid?(uidt(idt))(l, a) ≡ up?(from_byte(idt)(l, a)))
∧
(∀ (d: Data, a: Address): valid?(uidt(idt))(to_byte(idt)(d, a), a)) ∧
uninterpreted_data_type?(idt`uidt)
⊃
(∀ (d₁: Data, a₁: Address): up?[Data](from_byte(idt)(to_byte(idt)(d₁, a₁), a₁)))

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `interpreted_data_type?_TCC1`.
Q.E.D.

C.105.2 Interpreted_Data.pod_is_interpreted_data

Terse proof for `pod_is_interpreted_data`.
`pod_is_interpreted_data`:

```
{1}  ∀ (x: (pod_data_type?)): interpreted_data_type?(x)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pod_is_interpreted_data`.
Q.E.D.

C.105.3 Interpreted_Data.pod_size

Terse proof for `pod_size`.
`pod_size`:

```
{1}  ∀ (dt: (pod_data_type?)): size(uidt(dt)) > 0
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pod_data_type?,
which is trivially true.
This completes the proof of `pod_size`.
Q.E.D.

C.105.4 Interpreted_Data.data_type_length

Terse proof for `data_type_length`.
`data_type_length`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?), l: list⌊Byte⌋):
       valid?(uidt(idt))(l, addr) ⊃ length(l) = size(uidt(idt))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `data_type_length`.
Q.E.D.

C.105.5 Interpreted_Data.length_to_byte

Terse proof for `length_to_byte`.
`length_to_byte`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?), data: Data):
       length(to_byte(idt)(data, addr)) = size(uidt(idt))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of interpreted_data_type?,
Expanding the definition of uninterpreted_data_type?,

Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -1 with the terms: (to_byte(idt!1)(data!1, addr!1) addr!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
This completes the proof of `length_to_byte`.
Q.E.D.

C.105.6 Interpreted_Data.valid_from_byte

Terse proof for `valid_from_byte`.
`valid_from_byte`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?), l: list⌊Byte⌋):
       idt‘uidt‘valid?(l, addr) ⊃ up?(from_byte(idt)(l, addr))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_from_byte`.
Q.E.D.

C.105.7 Interpreted_Data.valid_iff_from_byte

Terse proof for `valid_iff_from_byte`.
`valid_iff_from_byte`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?), l: list⌊Byte⌋):
       idt‘uidt‘valid?(l, addr) ⇔ up?(from_byte(idt)(l, addr))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_iff_from_byte`.
Q.E.D.

C.105.8 Interpreted_Data.valid_to_byte

Terse proof for `valid_to_byte`.
`valid_to_byte`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?), data: Data):
       idt‘uidt‘valid?(to_byte(idt)(data, addr), addr)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `valid_to_byte`.
Q.E.D.

C.105.9 Interpreted_Data.up_from_byte_to_byte

Terse proof for `up_from_byte_to_byte`.
`up_from_byte_to_byte`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?), data: Data):
       up?(from_byte(idt)(to_byte(idt)(data, addr), addr))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of interpreted_data_type?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `up_from_byte_to_byte`.
Q.E.D.

C.105.10 Interpreted_Data.from_byte_to_byte

Terse proof for `from_byte_to_byte`.
`from_byte_to_byte`:

{1} ∀ (addr: Address, idt: (interpreted_data_type?), data: Data):
from_byte(idt)(to_byte(idt)(data, addr), addr) = up(data)

Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `from_byte_to_byte`.
Q.E.D.

C.105.11 Interpreted_Data.in_blessed_memory_subset

Terse proof for `in_blessed_memory_subset`.
`in_blessed_memory_subset`:

{1} ∀ (addr: Address, idt: (interpreted_data_type?), address_range: PRED⌊Address⌋):
in_blessed_memory?(idt, addr, address_range) ⊃
(address_block(addr, size(uidt(idt))) ⊆ address_range)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `in_blessed_memory_subset`.
Q.E.D.

C.106 Proofs for Interpreted_Data_Lift (abstract_data.pvs)

This theory contains no provable formal statements.

C.107 Proofs for IterationStatements1 (statements.pvs)

C.107.1 IterationStatements1.iterate_while_TCC1

Terse proof for `iterate_while_TCC1`.
`iterate_while_TCC1`:

{1} ∀ (n: nat, s: State): ¬ $n = 0$ ⊃ $n - 1 \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_while_TCC1`.
Q.E.D.

C.107.2 IterationStatements1.iterate_while_TCC2

Terse proof for `iterate_while_TCC2`.
`iterate_while_TCC2:`

```
{1}  ∀ (n: nat, s: State): ¬ n = 0 ⊃ n − 1 < n
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_while_TCC2`.
Q.E.D.

C.107.3 IterationStatements1.while_termination_point?_TCC1

Terse proof for `while_termination_point?_TCC1`.
`while_termination_point?_TCC1:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          s: State, n: nat, res: StmtResult[State]):
        OK?(res) ∧ res = iterate_while(n, b_ex, body)(s) ⊃
         OK?[State](res) ∨
          Break?[State](res) ∨
           Continue?[State](res) ∨
            Return?[State](res) ∨
             Switch?[State](res) ∨ Default?[State](res) ∨ Exception?[State](res)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_termination_point?_TCC1`.
Q.E.D.

C.107.4 IterationStatements1.while_TCC1

Terse proof for `while_TCC1`.
`while_TCC1:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          s: State):
        (∃ (n: nat): while_termination_point?(b_ex, body, s)(n)) ⊃
         nonempty?[nat](while_termination_point?(b_ex, body, s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_TCC1`.
Q.E.D.

C.107.5 IterationStatements1.nonzero_min_while_termination_point

Terse proof for `nonzero_min_while_termination_point`.

`nonzero_min_while_termination_point:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          s: State):
       (∃ (n: nat): while_termination_point?(b_ex, body, s)(n)) ∧
        ¬ min[nat](while_termination_point?(b_ex, body, s)) = 0
        ⊃ OK?(b_ex(s)) ∧ data(b_ex(s))
```

Repeatedly Skolemizing and flattening,

Rewriting using min_def, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `nonzero_min_while_termination_point`.

Q.E.D.

C.107.6 IterationStatements1.while_as_if_while

Terse proof for `while_as_if_while`.

`while_as_if_while:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]]):
       while(b_ex, body) =
        if_else(b_ex, body ## catch_continue ## while(b_ex, body) ## catch_break, skip)
```

Expanding the definition of while,

Expanding the definition of if_else,

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

we get 3 subgoals:

`while_as_if_while.1:`

```
{1}  IF ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
       THEN (iterate_while(min[nat](while_termination_point?(b_ex′, body′, x′)), b_ex′,
                           body′)
               ## e2s(b_ex′)
               ## catch_break)
                (x′)
     ELSE Hang
     ENDIF
      =
      (b_ex′ ##
         (λ (b: bool):
            IF b
              THEN (body′ ## catch_continue ##
                      (λ (s: State):
                         IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                           THEN (iterate_while(min[nat]
                                               (while_termination_point?(b_ex′,
                                                                         body′,
                                                                         s)),
                                               b_ex′, body′)
                                   ## e2s(b_ex′)
                                   ## catch_break)
                                    (s)
                         ELSE Hang
                         ENDIF))
                    ## catch_break
            ELSE skip
            ENDIF))
          (x′)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`while_as_if_while.1.1:`

```
{-1}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{1}   (iterate_while(min⌊nat⌋(while_termination_point?(b_ex′, body′, x′)), b_ex′, body′) ##
         e2s(b_ex′)
         ## catch_break)
          (x′)
       =
       (b_ex′ ##
          (λ (b: bool):
             IF b
               THEN (body′ ## catch_continue ##
                       (λ (s: State):
                          IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                            THEN (iterate_while(min⌊nat⌋
                                                (while_termination_point?(b_ex′,
                                                                          body′,
                                                                          s)),
                                                b_ex′, body′)
                                    ## e2s(b_ex′)
                                    ## catch_break)
                                     (s)
                          ELSE Hang
                          ENDIF))
                    ## catch_break
             ELSE skip
             ENDIF))
           (x′)
```

Case splitting on min(while_termination_point?(b_ex!1, body!1, x!1)) = 0,

we get 2 subgoals:

`while_as_if_while.1.1.1:`

```
{-1}  min(while_termination_point?(b_ex′, body′, x′)) = 0
{-2}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
------------------------------------------------------------------------
{1}   (iterate_while(min[nat](while_termination_point?(b_ex′, body′, x′)), b_ex′, body′) ##
         e2s(b_ex′)
         ## catch_break)
          (x′)
       =
       (b_ex′ ##
          (λ (b: bool):
               IF b
                 THEN (body′ ## catch_continue ##
                           (λ (s: State):
                                IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                                  THEN (iterate_while(min[nat]
                                                          (while_termination_point?(b_ex′,
                                                                                    body′
                                                                                    s)),
                                                      b_ex′, body′)
                                                 ## e2s(b_ex′)
                                                 ## catch_break)
                                                  (s)
                                  ELSE Hang
                                  ENDIF))
                       ## catch_break
               ELSE skip
               ENDIF))
            (x′)
```

Replacing using formula -1,

Rewriting using min_def, matching in *,

Expanding the definition of minimum?,

Expanding the definition of while_termination_point?,

Expanding the definition of iterate_while,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.1`.

`while_as_if_while.1.1.2:`

```
{-1}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
------------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{2}   (iterate_while(min[nat](while_termination_point?(b_ex′, body′, x′)), b_ex′, body′) ##
         e2s(b_ex′)
         ## catch_break)
          (x′)
       =
       (b_ex′ ##
          (λ (b: bool):
               IF b
                 THEN (body′ ## catch_continue ##
                           (λ (s: State):
                                IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                                  THEN (iterate_while(min[nat]
                                                      (while_termination_point?(b_ex′,
                                                                                body′,
                                                                                s)),
                                                      b_ex′, body′)
                                                ## e2s(b_ex′)
                                                ## catch_break)
                                                 (s)
                                ELSE Hang
                                ENDIF))
                       ## catch_break
                 ELSE skip
                 ENDIF))
            (x′)
```

Using lemma nonzero_min_while_termination_point,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of iterate_while,

Expanding the definition of ##,

Expanding the definition of lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`while_as_if_while.1.1.2.1:`

{-1} ∃ (n: nat): while_termination_point?(b_ex′, body′, x')(n)
{-2} OK?(b_ex′(x'))
{-3} data(b_ex′(x'))
{-4} OK?(e2s(b_ex′)(x'))

{1} min(while_termination_point?(b_ex′, body′, x')) = 0
{2} catch_break(lift(e2s(b_ex′))
(lift(iterate_while(min [nat](while_termination_point?(b_ex′, body′, x')) − 1,
b_ex′, body′))
(catch_continue(body′(state(e2s(b_ex′)(x')))))))
=
catch_break(lift(λ (s: State):
IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
THEN catch_break(lift(e2s(b_ex′))
(iterate_while(min[nat]
(while_termination_poin
(b_ex′, body′, s)),
b_ex′,
body′)
(s)))
ELSE Hang
ENDIF)
(catch_continue(body′(state(b_ex′(x'))))))

Expanding the definition of e2s,

Expanding the definition of lift,

Case splitting on OK?(catch_continue(body!1(state(b_ex!1(x!1))))),

we get 2 subgoals:

`while_as_if_while.1.1.2.1.1:`

```
{-1}  OK?(catch_continue(body′(state(b_ex′(x′)))))
{-2}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-3}  OK?(b_ex′(x′))
{-4}  data(b_ex′(x′))
{-5}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{2}   catch_break(CASES CASES catch_continue(body′(state(b_ex′(x′)))) OF
                              OK(state_1):
                                iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) − 1,
                                              b_ex′, body′)
                                             (state_1)
                              ELSE catch_continue(body′(state(b_ex′(x′))))
                              ENDCASES OF
                        OK(state_1): e2s(b_ex′)(state_1)
                        ELSE CASES catch_continue(body′(state(b_ex′(x′)))) OF
                                OK(state_1):
                                  iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) − 1,
                                                b_ex′, body′)
                                               (state_1)
                                ELSE catch_continue(body′(state(b_ex′(x′))))
                                ENDCASES
                        ENDCASES)
      =
      catch_break(CASES catch_continue(body′(state(b_ex′(x′)))) OF
                        OK(state_1):
                          IF ∃ (n: nat): while_termination_point?(b_ex′, body′, state_1)(n)
                            THEN catch_break(CASES iterate_while(min[nat]
                                                                   (while_termination_point?
                                                                     (b_ex′, body′, state_1)),
                                                                 b_ex′, body′)
                                                                (state_1) OF
                                                   OK(state_1): e2s(b_ex′)(state_1)
                                                   ELSE iterate_while(min[nat]
                                                                        (while_termination_point?
                                                                          (b_ex′, body′, state_1)),
                                                                      b_ex′,
                                                                      body′)
                                                                     (state_1)
                                                   ENDCASES)
                            ELSE Hang
                            ENDIF
                        ELSE catch_continue(body′(state(b_ex′(x′))))
                        ENDCASES)
```

Simplifying, rewriting, and recording with decision procedures,

Case splitting on EXISTS (n: nat): while_termination_point?(b_ex!1, body!1, state(catch_continue (body!1(state(b_ex!1(x!1)))))) (n),

we get 2 subgoals:

```
while_as_if_while.1.1.2.1.1.1:

{-1}  ∃ (n: nat):
        while_termination_point?(b_ex′, body′,
                                 state(catch_continue(body′(state(b_ex′(x′))))))
                                (n)
{-2}  OK?(catch_continue(body′(state(b_ex′(x′)))))
{-3}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-4}  OK?(b_ex′(x′))
{-5}  data(b_ex′(x′))
{-6}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{2}   catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) - 1,
                                       b_ex′, body′)
                                      (state(catch_continue(body′(state(b_ex′(x′))))))) OF
                    OK(state_1): e2s(b_ex′)(state_1)
                    ELSE CASES catch_continue(body′(state(b_ex′(x′)))) OF
                               OK(state_1):
                                 iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) -
                                               b_ex′, body′)
                                              (state_1)
                               ELSE catch_continue(body′(state(b_ex′(x′))))
                               ENDCASES
                    ENDCASES)
        =
        IF ∃ (n: nat):
             while_termination_point?(b_ex′, body′,
                                      state(catch_continue(body′(state(b_ex′(x′))))))
                                     (n)
          THEN catch_break(catch_break(CASES iterate_while(min[nat]
                                                           (while_termination_point?
                                                              (b_ex′,
                                                                 body′,
                                                                 state
                                                                 (catch_continue
                                                                    (body′
                                                                       (state
                                                                          (b_ex′(x′)))))),
                                                           b_ex′, body′)
                                                          (state(catch_continue
                                                                   (body′
                                                                      (state(b_ex′(x′))))) O
                                       OK(state_1): e2s(b_ex′)(state_1)
                                       ELSE iterate_while(min[nat]
                                                          (while_termination_point?
                                                             (b_ex′,
                                                                body′,
                                                                state
                                                                (catch_continue
                                                                   (body′
                                                                      (state
                                                                         (b_ex′(x′)))))),
                                                          b_ex′, body′)
                                                         (state(catch_continue
                                                                  (body′
                                                                     (state(b_ex′(x′))))))
                                       ENDCASES))
        ELSE catch_break(Hang)
        ENDIF
```

Replacing using formula -1,

Case splitting on min(while_termination_point?(b_ex!1, body!1, x!1)) - 1 = min(while_termination_point?(b_ex!1, body!1, state(catch_continue (body!1(state(b_ex!1(x!1)))))))),

we get 2 subgoals:

```
while_as_if_while.1.1.2.1.1.1.1:
```

```
{-1}  min(while_termination_point?(b_ex′, body′, x′)) − 1 =
        min(while_termination_point?(b_ex′, body′,
                                     state(catch_continue(body′(state(b_ex′(x′)))))))
{-2}  ∃ (n: nat):
        while_termination_point?(b_ex′, body′,
                                 state(catch_continue(body′(state(b_ex′(x′))))))
                                (n)
{-3}  OK?(catch_continue(body′(state(b_ex′(x′)))))
{-4}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-5}  OK?(b_ex′(x′))
{-6}  data(b_ex′(x′))
{-7}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{2}   catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) − 1,
                                       b_ex′, body′)
                                      (state(catch_continue(body′(state(b_ex′(x′)))))) OF
                    OK(state_1): e2s(b_ex′)(state_1)
                    ELSE CASES catch_continue(body′(state(b_ex′(x′)))) OF
                              OK(state_1):
                                iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′))
                                              b_ex′, body′)
                                             (state_1)
                              ELSE catch_continue(body′(state(b_ex′(x′))))
                              ENDCASES
                    ENDCASES)
      =
      catch_break(catch_break(CASES iterate_while(min[nat]
                                                  (while_termination_point?(b_ex′,
                                                                            boo
                                                                            sta
                                                                            (ca
                                                                              (
                                                                                b_ex′, body′)
                                                                               (state(catch_continue(body′
                                                                                                     (state
                                                                                                       (b_ex′
                                                                                                         (x′))))))
                                                  OK(state_1): e2s(b_ex′)(state_1)
                                                  ELSE iterate_while(min[nat]
                                                                     (while_termination_point?(b_ex′,
                                                                                               bo
                                                                                               st
                                                                                               (c
                                                                                                 b_ex′, body′)
                                                                                                (state(catch_continue(body′
                                                                                                                      (state
                                                                                                                        (b_ex′(x′))
                                                  ENDCASES))
```

Replacing using formula -1,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.1.1`.

```
while_as_if_while.1.1.2.1.1.1.2:
```

```
{-1}  ∃ (n: nat):
        while_termination_point?(b_ex′, body′,
                                 state(catch_continue(body′(state(b_ex′(x′))))))
                                (n)
{-2}  OK?(catch_continue(body′(state(b_ex′(x′)))))
{-3}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-4}  OK?(b_ex′(x′))
{-5}  data(b_ex′(x′))
{-6}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) − 1 =
       min(while_termination_point?(b_ex′, body′,
                                    state(catch_continue(body′(state(b_ex′(x′)))))))
{2}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{3}   catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) − 1,
                                       b_ex′, body′)
                                      (state(catch_continue(body′(state(b_ex′(x′)))))) OF
                    OK(state_1): e2s(b_ex′)(state_1)
                    ELSE CASES catch_continue(body′(state(b_ex′(x′)))) OF
                              OK(state_1):
                                iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) −
                                              b_ex′, body′)
                                             (state_1)
                              ELSE catch_continue(body′(state(b_ex′(x′))))
                              ENDCASES
                    ENDCASES)
      =
      catch_break(catch_break(CASES iterate_while(min[nat]
                                                  (while_termination_point?(b_ex′,
                                                                            boo
                                                                            sta
                                                                            (ca
                                                                              (
                                                                                b_ex′, body′)
                                                                               (state(catch_continue(body′
                                                                                                     (state
                                                                                                       (b_ex′
                                                                                                         (x′))))))
                                                  OK(state_1): e2s(b_ex′)(state_1)
                                                  ELSE iterate_while(min[nat]
                                                                     (while_termination_point?(b_ex′,
                                                                                               bo
                                                                                               st
                                                                                               (c
                                                                                                 b_ex′, body′)
                                                                                                (state(catch_continue(body′
                                                                                                                      (state
                                                                                                                        (b_ex′(x′))
                                                  ENDCASES))
```

Hiding formulas: 3,

Rewriting using min_plus1, matching in *,

we get 2 subgoals:

`while_as_if_while.1.1.2.1.1.1.2.1:`

{-1} $\exists$ (n: nat):
while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x'))))))
(n)
{-2} OK?(catch_continue(body′(state(b_ex′(x')))))
{-3} $\exists$ (n: nat): while_termination_point?(b_ex′, body′, x')(n)
{-4} OK?(b_ex′(x'))
{-5} data(b_ex′(x'))
{-6} OK?(e2s(b_ex′)(x'))

{1} $1 + \min(\lambda\ (n:\ \text{nat}):\ \text{while\_termination\_point?}(\text{b\_ex}',\ \text{body}',\ x')(1+n)) - 1$
=
min(while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x')))))))
{2} $1 + \min(\lambda\ (n:\ \text{nat}):\ \text{while\_termination\_point?}(\text{b\_ex}',\ \text{body}',\ x')(1+n)) = 0$

Case splitting on (LAMBDA (n: nat): while_termination_point?(b_ex!1, body!1, x!1)(1 + n)) = while_termination_point?(b_ex!1, body!1, state(catch_continue (body!1(state(b_ex!1(x!1)))))),

we get 2 subgoals:

`while_as_if_while.1.1.2.1.1.1.2.1.1:`

{-1} (λ (n: nat): while_termination_point?(b_ex′, body′, x')(1 + n)) =
while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x'))))))
{-2} $\exists$ (n: nat):
while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x'))))))
(n)
{-3} OK?(catch_continue(body′(state(b_ex′(x')))))
{-4} $\exists$ (n: nat): while_termination_point?(b_ex′, body′, x')(n)
{-5} OK?(b_ex′(x'))
{-6} data(b_ex′(x'))
{-7} OK?(e2s(b_ex′)(x'))

{1} $1 + \min(\lambda\ (n:\ \text{nat}):\ \text{while\_termination\_point?}(\text{b\_ex}',\ \text{body}',\ x')(1+n)) - 1$
=
min(while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x')))))))
{2} $1 + \min(\lambda\ (n:\ \text{nat}):\ \text{while\_termination\_point?}(\text{b\_ex}',\ \text{body}',\ x')(1+n)) = 0$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.1.2.1.1`.

`while_as_if_while.1.1.2.1.1.1.2.1.2:`

```
{-1}  ∃ (n: nat):
        while_termination_point?(b_ex′, body′,
                                 state(catch_continue(body′(state(b_ex′(x′))))))
                                (n)
{-2}  OK?(catch_continue(body′(state(b_ex′(x′)))))
{-3}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-4}  OK?(b_ex′(x′))
{-5}  data(b_ex′(x′))
{-6}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   (λ (n: nat): while_termination_point?(b_ex′, body′, x′)(1 + n)) =
       while_termination_point?(b_ex′, body′,
                                state(catch_continue(body′(state(b_ex′(x′))))))
{2}   1 + min(λ (n : nat) : while_termination_point?(b_ex′, body′, x′)(1 + n)) − 1
       =
       min(while_termination_point?(b_ex′, body′,
                                    state(catch_continue(body′(state(b_ex′(x′)))))))
{3}   1 + min(λ (n : nat) : while_termination_point?(b_ex′, body′, x′)(1 + n)) = 0
```

Hiding formulas: (2 3),

Applying decompose-equality,

Expanding the definition of while_termination_point?,

Expanding the definition of iterate_while,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.1.2.1.2`.

`while_as_if_while.1.1.2.1.1.1.2.2:`

```
{-1}  while_termination_point?(b_ex′, body′, x′)(0)
{-2}  ∃ (n: nat):
        while_termination_point?(b_ex′, body′,
                                 state(catch_continue(body′(state(b_ex′(x′))))))
                                (n)
{-3}  OK?(catch_continue(body′(state(b_ex′(x′)))))
{-4}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-5}  OK?(b_ex′(x′))
{-6}  data(b_ex′(x′))
{-7}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) − 1 =
       min(while_termination_point?(b_ex′, body′,
                                    state(catch_continue(body′(state(b_ex′(x′)))))))
{2}   min(while_termination_point?(b_ex′, body′, x′)) = 0
```

Hiding formulas: 1,

Rewriting using min_def, matching in *,

Expanding the definition of minimum?,

which is trivially true.

This completes the proof of `while_as_if_while.1.1.2.1.1.1.2.2`.

```
while_as_if_while.1.1.2.1.1.2:
```

{-1} OK?(catch_continue(body′(state(b_ex′(x′)))))
{-2} ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-3} OK?(b_ex′(x′))
{-4} data(b_ex′(x′))
{-5} OK?(e2s(b_ex′)(x′))

{1} ∃ (n: nat):
  while_termination_point?(b_ex′, body′,
    state(catch_continue(body′(state(b_ex′(x′))))))
  (n)

{2} min(while_termination_point?(b_ex′, body′, x′)) = 0

{3} catch_break(CASES iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) − 1,
    b_ex′, body′)
  (state(catch_continue(body′(state(b_ex′(x′))))))) OF
  OK(state_1): e2s(b_ex′)(state_1)
  ELSE CASES catch_continue(body′(state(b_ex′(x′)))) OF
    OK(state_1):
      iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) − 1,
        b_ex′, body′)
      (state_1)
    ELSE catch_continue(body′(state(b_ex′(x′))))
    ENDCASES
  ENDCASES)

=

IF ∃ (n: nat):
  while_termination_point?(b_ex′, body′,
    state(catch_continue(body′(state(b_ex′(x′))))))
  (n)
THEN catch_break(catch_break(CASES iterate_while(min[nat]
  (while_termination_point?
    (b_ex′,
      body′,
      state
      (catch_continue
        (body′
          (state
            (b_ex′(x′)))))))),
  b_ex′, body′)
(state(catch_continue
  (body′
    (state(b_ex′(x′))))))) OF
  OK(state_1): e2s(b_ex′)(state_1)
  ELSE iterate_while(min[nat]
    (while_termination_point?
      (b_ex′,
        body′,
        state
        (catch_continue
          (body′
            (state
              (b_ex′(x′)))))))),
    b_ex′, body′)
  (state(catch_continue
    (body′
      (state(b_ex′(x′))))))
  ENDCASES))
ELSE catch_break(Hang)
ENDIF

Hiding formulas: 3,
Repeatedly Skolemizing and flattening,
Case splitting on n!1 = 0,
we get 2 subgoals:
`while_as_if_while.1.1.2.1.1.2.1:`

{-1} $n' = 0$
{-2} $n' \geq 0$
{-3} OK?(catch_continue(body′(state(b_ex′(x')))))
{-4} while_termination_point?(b_ex′, body′, x')(n')
{-5} OK?(b_ex′(x'))
{-6} data(b_ex′(x'))
{-7} OK?(e2s(b_ex′)(x'))

{1} $\exists$ (n: nat):
while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x'))))))
(n)
{2} min(while_termination_point?(b_ex′, body′, x')) = 0

Replacing using formula -1,
Rewriting using min_def, matching in *,
Expanding the definition of minimum?,
which is trivially true.
This completes the proof of `while_as_if_while.1.1.2.1.1.2.1`.
`while_as_if_while.1.1.2.1.1.2.2:`

{-1} $n' \geq 0$
{-2} OK?(catch_continue(body′(state(b_ex′(x')))))
{-3} while_termination_point?(b_ex′, body′, x')(n')
{-4} OK?(b_ex′(x'))
{-5} data(b_ex′(x'))
{-6} OK?(e2s(b_ex′)(x'))

{1} $n' = 0$
{2} $\exists$ (n: nat):
while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x'))))))
(n)
{3} min(while_termination_point?(b_ex′, body′, x')) = 0

Instantiating the top quantifier in 2 with the terms: $n' - 1$,
we get 2 subgoals:
`while_as_if_while.1.1.2.1.1.2.2.1:`

{-1} $n' \geq 0$
{-2} OK?(catch_continue(body′(state(b_ex′(x')))))
{-3} while_termination_point?(b_ex′, body′, x')(n')
{-4} OK?(b_ex′(x'))
{-5} data(b_ex′(x'))
{-6} OK?(e2s(b_ex′)(x'))

{1} $n' = 0$
{2} while_termination_point?(b_ex′, body′,
state(catch_continue(body′(state(b_ex′(x'))))))
($n' - 1$)
{3} min(while_termination_point?(b_ex′, body′, x')) = 0

Hiding formulas: 3,

Expanding the definition of while_termination_point?,

Expanding the definition of iterate_while,

Applying disjunctive simplification to flatten sequent,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.2.2.1`.

`while_as_if_while.1.1.2.1.1.2.2.2`:

{-1}	$n' \geq 0$
{-2}	OK?(catch_continue(body′(state(b_ex′(x')))))
{-3}	while_termination_point?(b_ex′, body′, x')(n')
{-4}	OK?(b_ex′(x'))
{-5}	data(b_ex′(x'))
{-6}	OK?(e2s(b_ex′)(x'))
{1}	$n' - 1 \geq 0$
{2}	$n' = 0$
{3}	min(while_termination_point?(b_ex′, body′, x')) $= 0$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `while_as_if_while.1.1.2.1.1.2.2.2`.

`while_as_if_while.1.1.2.1.2:`

```
{-1}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-2}  OK?(b_ex′(x′))
{-3}  data(b_ex′(x′))
{-4}  OK?(e2s(b_ex′)(x′))
------------------------------------------------------------------------
{1}   OK?(catch_continue(body′(state(b_ex′(x′)))))
{2}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{3}   catch_break(CASES CASES catch_continue(body′(state(b_ex′(x′)))) OF
                    OK(state_1):
                      iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) −
                                    b_ex′, body′)
                                   (state_1)
                    ELSE catch_continue(body′(state(b_ex′(x′))))
                    ENDCASES OF
                  OK(state_1): e2s(b_ex′)(state_1)
                  ELSE CASES catch_continue(body′(state(b_ex′(x′)))) OF
                    OK(state_1):
                      iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) −
                                    b_ex′, body′)
                                   (state_1)
                    ELSE catch_continue(body′(state(b_ex′(x′))))
                    ENDCASES
                  ENDCASES)
      =
      catch_break(CASES catch_continue(body′(state(b_ex′(x′)))) OF
                  OK(state_1):
                    IF ∃ (n: nat): while_termination_point?(b_ex′, body′, state_1)(n)
                      THEN catch_break(CASES iterate_while(min[nat]
                                                           (while_termination_poin
                                                             (b_ex′, body′, state
                                                           b_ex′, body′)
                                                          (state_1) OF
                                         OK(state_1): e2s(b_ex′)(state_1)
                                         ELSE iterate_while(min[nat]
                                                            (while_termination_point?
                                                              (b_ex′, body′, state_1))
                                                            b_ex′,
                                                            body′)
                                                           (state_1)
                                         ENDCASES)
                      ELSE Hang
                      ENDIF
                  ELSE catch_continue(body′(state(b_ex′(x′))))
                  ENDCASES)
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `while_as_if_while.1.1.2.1.2`.

`while_as_if_while.1.1.2.2:`

```
{-1}  ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{-2}  OK?(b_ex′(x′))
{-3}  data(b_ex′(x′))
------------------------------------------------------------------------
{1}   min(while_termination_point?(b_ex′, body′, x′)) = 0
{2}   OK?(e2s(b_ex′)(x′))
{3}   catch_break(lift(e2s(b_ex′))
                       (lift(iterate_while(min [nat](while_termination_point?(b_ex′, body′, x′)) - 1,
                                           b_ex′, body′))
                            (catch_continue(e2s(b_ex′)(x′)))))
      =
      catch_break(lift(λ (s: State):
                         IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                           THEN catch_break(lift(e2s(b_ex′))
                                                 (iterate_while(min[nat]
                                                                (while_termination_point?
                                                                  (b_ex′, body′, s)),
                                                                b_ex′,
                                                                body′)
                                                               (s)))
                         ELSE Hang
                         ENDIF)
                       (catch_continue(body′(state(b_ex′(x′)))))))
```

Expanding the definition of e2s,
which is trivially true.
This completes the proof of `while_as_if_while.1.1.2.2`.
`while_as_if_while.1.2:`

```
------------------------------------------------------------------------
{1}   ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{2}   Hang?((b_ex′ ##
               (λ (b: bool):
                  IF b
                    THEN (body′ ## catch_continue ##
                                (λ (s: State):
                                   IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                                     THEN (iterate_while(min[nat]
                                                         (while_termination_point?(b_ex′,
                                                                                   body′,
                                                                                   s)),
                                                         b_ex′, body′)
                                                  ## e2s(b_ex′)
                                                  ## catch_break)
                                                 (s)
                                   ELSE Hang
                                   ENDIF))
                                ## catch_break
                  ELSE skip
                  ENDIF))
               (x′))
```

Copying formula number: 1
Copying formula number: 1

Instantiating the top quantifier in 1 with the terms: 0,
Instantiating the top quantifier in 2 with the terms: 1,
Expanding the definition of while_termination_point?,
Expanding the definition of iterate_while,
Expanding the definition of iterate_while,
Expanding the definition of skip,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Case splitting on EXISTS (n: nat): while_termination_point?(b_ex!1, body!1, state((e2s(b_ex!1) ## body!1 ## catch_continue ## (LAMBDA (s): OK(s))) (x!1))) (n),
we get 2 subgoals:
`while_as_if_while.1.2.1:`

```
{-1}  ∃ (n: nat):
        while_termination_point?(b_ex′, body′,
                                 state((e2s(b_ex′) ## body′ ## catch_continue ##
                                        (λ (s): OK(s)))
                                       (x′)))
                                (n)
{-2}  OK?(b_ex′(x′))
{-3}  data(b_ex′(x′))
{-4}  OK?((e2s(b_ex′) ## body′ ## catch_continue ## (λ (s): OK(s)))(x′))
{-5}  OK?(b_ex′(state((e2s(b_ex′) ## body′ ## catch_continue ## (λ (s): OK(s)))
                      (x′))))
{-6}  data(b_ex′(state((e2s(b_ex′) ## body′ ## catch_continue ## (λ (s): OK(s)))
                      (x′))))
------------------------------------------------------------------------------
{1}   ∃ (n: nat): while_termination_point?(b_ex′, body′, x′)(n)
{2}   Hang?((b_ex′ ##
             (λ (b: bool):
                IF b
                  THEN (body′ ## catch_continue ##
                          (λ (s: State):
                             IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                               THEN (iterate_while(min[nat]
                                                   (while_termination_point?(b_ex′
                                                                             b_ex′, body′)
                                                    ## e2s(b_ex′)
                                                    ## catch_break)
                                                   (s)
                                     ELSE Hang
                                     ENDIF))
                          ## catch_break
                  ELSE λ (s: State): OK(s)
                  ENDIF))
              (x′))
```

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: $(n' + 1)$,
Expanding the definition of while_termination_point?,
Expanding the definition of iterate_while,
Applying disjunctive simplification to flatten sequent,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_as_if_while.1.2.1`.

`while_as_if_while.1.2.2:`

```
{-1}  OK?(b_ex'(x'))
{-2}  data(b_ex'(x'))
{-3}  OK?((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)))(x'))
{-4}  OK?(b_ex'(state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)))
                      (x'))))
{-5}  data(b_ex'(state((e2s(b_ex') ## body' ## catch_continue ## (λ (s): OK(s)))
                       (x'))))
-------------------------------------------------------------------------------
{1}   ∃ (n: nat):
        while_termination_point?(b_ex', body',
                                 state((e2s(b_ex') ## body' ## catch_continue ##
                                        (λ (s): OK(s)))
                                       (x')))
                                (n)
{2}   ∃ (n: nat): while_termination_point?(b_ex', body', x')(n)
{3}   Hang?((b_ex' ##
               (λ (b: bool):
                  IF b
                    THEN (body' ## catch_continue ##
                            (λ (s: State):
                               IF ∃ (n: nat): while_termination_point?(b_ex', body', s)(n)
                                 THEN (iterate_while(min[nat]
                                                     (while_termination_point?(b_ex',
                                                                               body',
                                                                               s)),
                                                     b_ex', body')
                                       ## e2s(b_ex')
                                       ## catch_break)
                                      (s)
                                 ELSE Hang
                                 ENDIF))
                          ## catch_break
                    ELSE λ (s: State): OK(s)
                    ENDIF))
             (x'))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.1.2.2`.

`while_as_if_while.2:`

```
-------------------------------------------------------------------------------
{1}   ∀ (b: bool):
        b ⊃
          (∀ (s: State):
             (∃ (n: nat): while_termination_point?(b_ex', body', s)(n)) ⊃
               nonempty?[nat](while_termination_point?(b_ex', body', s)))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `while_as_if_while.2`.

`while_as_if_while.3:`

{1} ∀ (s: State): ($\exists$ (n: nat): while_termination_point?(b_ex′, body′, s)(n)) ⊃ nonempty?[nat](while_termination_point?(b_ex′, body′, s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_as_if_while.3`.
Q.E.D.

C.107.7 IterationStatements1.iterate_do_as_while

Terse proof for `iterate_do_as_while`.
`iterate_do_as_while:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], n: nat): iterate_do(n, body, b_ex) = (body ## catch_continue ## iterate_while(n, b_ex, body))

Inducting on n on formula 1,
we get 2 subgoals:
`iterate_do_as_while.1:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]]): iterate_do(0, body, b_ex) = (body ## catch_continue ## iterate_while(0, b_ex, body))

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
Applying decompose-equality,
This completes the proof of `iterate_do_as_while.1`.
`iterate_do_as_while.2:`

{1} ∀ j: (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]]): iterate_do(j, body, b_ex) = (body ## catch_continue ## iterate_while(j, b_ex, body))) ⊃ (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]]): iterate_do(j + 1, body, b_ex) = (body ## catch_continue ## iterate_while(j + 1, b_ex, body)))

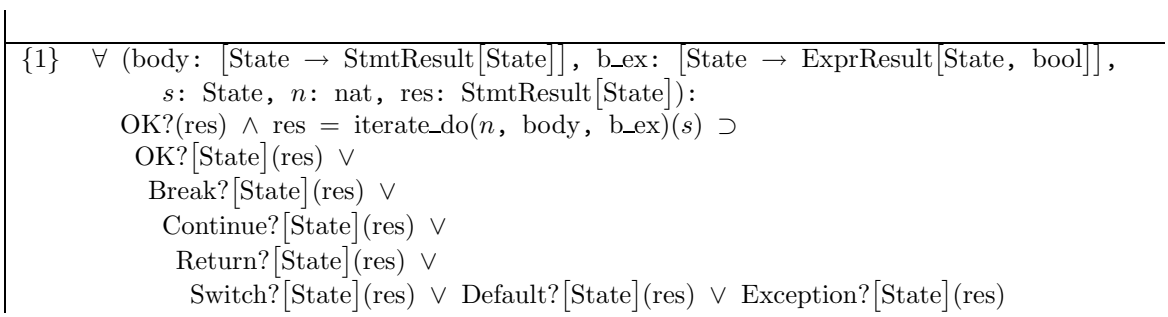
Repeatedly Skolemizing and flattening,
Expanding the definition of iterate_do,
Expanding the definition of iterate_while,
Applying decompose-equality,
Instantiating quantified variables,
Replacing using formula -2,

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_do_as_while.2`.
Q.E.D.

C.107.8 IterationStatements1.do_termination_point?_TCC1

Terse proof for `do_termination_point?_TCC1`.

`do_termination_point?_TCC1:`

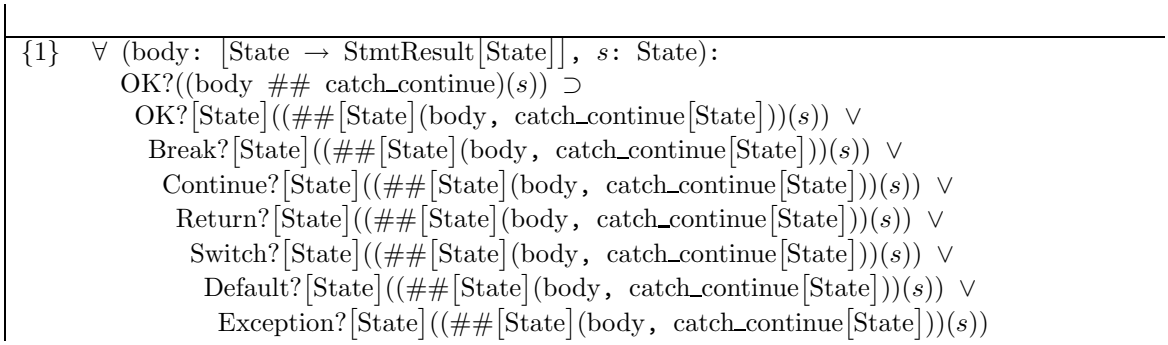


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `do_termination_point?_TCC1`.
Q.E.D.

C.107.9 IterationStatements1.termination_point_do_as_while_TCC1

Terse proof for `termination_point_do_as_while_TCC1`.

`termination_point_do_as_while_TCC1:`

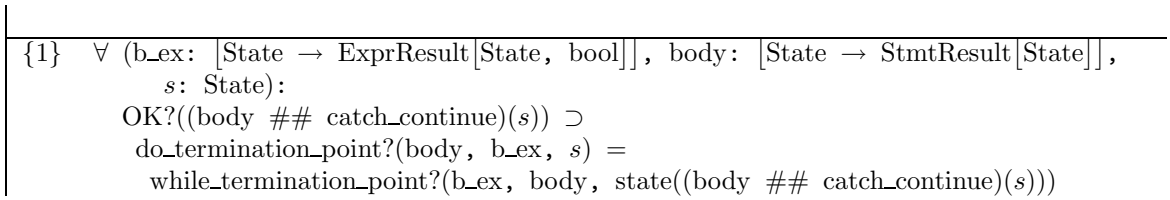


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `termination_point_do_as_while_TCC1`.
Q.E.D.

C.107.10 IterationStatements1.termination_point_do_as_while

Terse proof for `termination_point_do_as_while`.

termination_point_do_as_while:



Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Expanding the definition of do_termination_point?,

Expanding the definition of while_termination_point?,

Rewriting using iterate_do_as_while, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

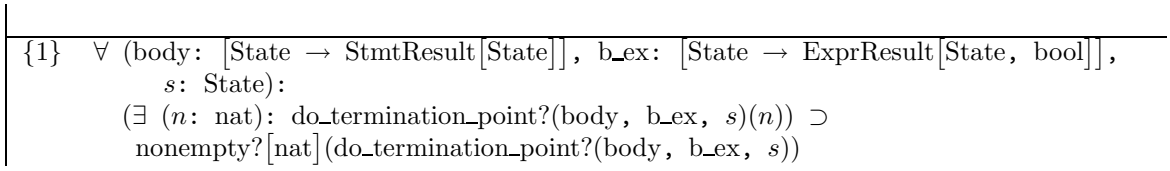
This completes the proof of termination_point_do_as_while.

Q.E.D.

C.107.11 IterationStatements1.do_while_TCC1

Terse proof for do_while_TCC1.

do_while_TCC1:



Trying repeated skolemization, instantiation, and if-lifting,

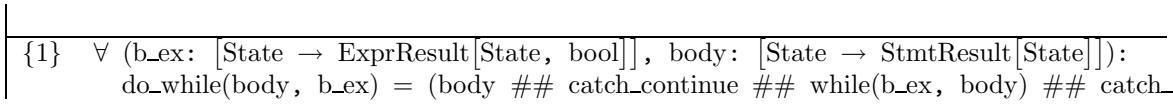
This completes the proof of do_while_TCC1.

Q.E.D.

C.107.12 IterationStatements1.do_as_while

Terse proof for do_as_while.

do_as_while:



Repeatedly Skolemizing and flattening,

Expanding the definition of do_while,

Expanding the definition of while,

Applying decompose-equality,

we get 3 subgoals:

`do_as_while.1:`

```
{1}  IF ∃ (n: nat): do_termination_point?(body′, b_ex′, x′)(n)
       THEN (iterate_do(min[nat](do_termination_point?(body′, b_ex′, x′)), body′, b_ex′)
               ## e2s(b_ex′)
               ## catch_break)
                (x′)
     ELSE Hang
     ENDIF
      =
     ((body′ ## catch_continue ##
           (λ (s: State):
               IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                 THEN (iterate_while(min[nat](while_termination_point?(b_ex′, body′, s)),
                                     b_ex′, body′)
                             ## e2s(b_ex′)
                             ## catch_break)
                              (s)
               ELSE Hang
               ENDIF))
        ## catch_break)
         (x′)
```

Installing automatic rewrites from: (iterate_do_as_while! termination_point_do_as_while!)

Case splitting on OK?((body!1 ## catch_continue)(x!1)),

we get 2 subgoals:

`do_as_while.1.1:`

```
{-1}  OK?((body′ ## catch_continue)(x′))
{1}  IF ∃ (n: nat): do_termination_point?(body′, b_ex′, x′)(n)
       THEN (iterate_do(min[nat](do_termination_point?(body′, b_ex′, x′)), body′, b_ex′)
               ## e2s(b_ex′)
               ## catch_break)
                (x′)
     ELSE Hang
     ENDIF
      =
     ((body′ ## catch_continue ##
           (λ (s: State):
               IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
                 THEN (iterate_while(min[nat](while_termination_point?(b_ex′, body′, s)),
                                     b_ex′, body′)
                             ## e2s(b_ex′)
                             ## catch_break)
                              (s)
               ELSE Hang
               ENDIF))
        ## catch_break)
         (x′)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `do_as_while.1.1`.

`do_as_while.1.2:`

{1} OK?((body′ ## catch_continue)(x′))
{2} IF ∃ (n: nat): do_termination_point?(body′, b_ex′, x′)(n)
THEN (iterate_do(min[nat](do_termination_point?(body′, b_ex′, x′)), body′, b_ex′)
e2s(b_ex′)
catch_break)
(x′)
ELSE Hang
ENDIF
=
((body′ ## catch_continue ##
(λ (s: State):
IF ∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)
THEN (iterate_while(min[nat](while_termination_point?(b_ex′, body′, s)),
b_ex′, body′)
e2s(b_ex′)
catch_break)
(s)
ELSE Hang
ENDIF))
catch_break)
(x′)

Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:
`do_as_while.1.2.1:`

{-1} Break?(body′(x′))
{1} ∃ (n: nat): do_termination_point?(body′, b_ex′, x′)(n)

Instantiating the top quantifier in 1 with the terms: 0,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `do_as_while.1.2.1`.
`do_as_while.1.2.2:`

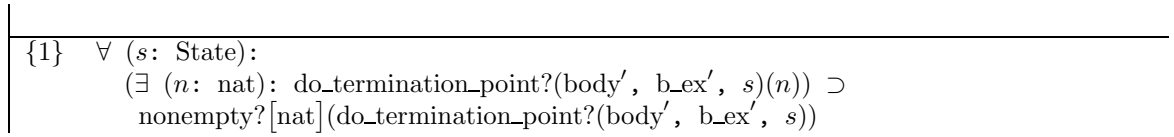
{1} Continue?(body′(x′))
{2} OK?(body′(x′))
{3} ∃ (n: nat): do_termination_point?(body′, b_ex′, x′)(n)
{4} Hang?(body′(x′))

Instantiating the top quantifier in 3 with the terms: 0,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `do_as_while.1.2.2`.
`do_as_while.2:`

{1} ∀ (s: State):
(∃ (n: nat): while_termination_point?(b_ex′, body′, s)(n)) ⊃
nonempty?[nat](while_termination_point?(b_ex′, body′, s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `do_as_while.2`.

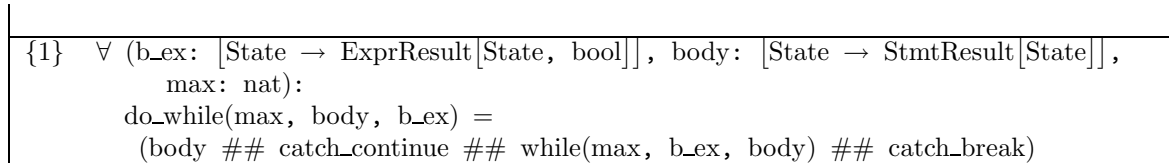
do_as_while.3:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of do_as_while.3.
Q.E.D.

C.107.13 IterationStatements1.do_while_unroll

Terse proof for do_while_unroll.
do_while_unroll:

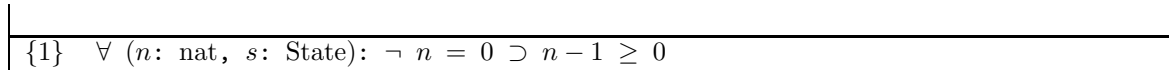


Repeatedly Skolemizing and flattening,
Expanding the definition of do_while,
Expanding the definition of while,
Rewriting using do_as_while, matching in *,
This completes the proof of do_while_unroll.
Q.E.D.

C.108 Proofs for IterationStatements2 (statements.pvs)

C.108.1 IterationStatements2.iterate_for_TCC1

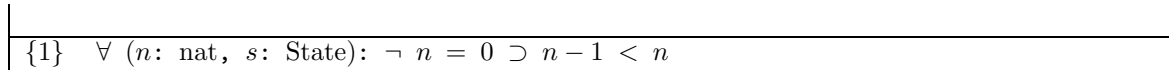
Terse proof for iterate_for_TCC1.
iterate_for_TCC1:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of iterate_for_TCC1.
Q.E.D.

C.108.2 IterationStatements2.iterate_for_TCC2

Terse proof for iterate_for_TCC2.
iterate_for_TCC2:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of iterate_for_TCC2.
Q.E.D.

C.108.3 IterationStatements2.iterate_for_as_while_TCC1

Terse proof for `iterate_for_as_while_TCC1`.

`iterate_for_as_while_TCC1:`

$$\{1\}\quad \forall\ (n\colon \text{nat})\colon\ n > 0 \supset n - 1 \geq 0$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `iterate_for_as_while_TCC1`.

Q.E.D.

C.108.4 IterationStatements2.iterate_for_as_while

Terse proof for `iterate_for_as_while`.

`iterate_for_as_while:`

$$\begin{array}{l}\{1\}\quad \forall\ (\text{b\_ex}\colon [\text{State} \to \text{ExprResult}[\text{State}, \text{bool}]],\ \text{body}\colon [\text{State} \to \text{StmtResult}[\text{State}]],\\ \qquad\quad \text{expr}\colon [\text{State} \to \text{ExprResult}[\text{State}, \text{ExprData}]],\ n\colon \text{nat})\colon\\ \qquad n > 0 \supset\\ \qquad\ \text{iterate\_for}(n, \text{b\_ex}, \text{expr}, \text{body}) =\\ \qquad\quad (\text{e2s}(\text{b\_ex}) \mathbin{\#\#} \text{body} \mathbin{\#\#} \text{catch\_continue} \mathbin{\#\#} \text{iterate\_while}(n-1, \text{expr} \mathbin{\#\#} \text{b\_ex}, \text{body})\\ \qquad\qquad \mathbin{\#\#} \text{e2s}(\text{expr}))\end{array}$$

Inducting on n on formula 1,

we get 3 subgoals:

`iterate_for_as_while.1:`

$$\begin{array}{l}\{1\}\quad \forall\ (\text{b\_ex}\colon [\text{State} \to \text{ExprResult}[\text{State}, \text{bool}]],\ \text{body}\colon [\text{State} \to \text{StmtResult}[\text{State}]],\\ \qquad\quad \text{expr}\colon [\text{State} \to \text{ExprResult}[\text{State}, \text{ExprData}]])\colon\\ \qquad 0 > 0 \supset\\ \qquad\ \text{iterate\_for}(0, \text{b\_ex}, \text{expr}, \text{body}) =\\ \qquad\quad (\text{e2s}(\text{b\_ex}) \mathbin{\#\#} \text{body} \mathbin{\#\#} \text{catch\_continue} \mathbin{\#\#} \text{iterate\_while}(0-1, \text{expr} \mathbin{\#\#} \text{b\_ex}, \text{body}) \#\\ \qquad\qquad \text{e2s}(\text{expr}))\end{array}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `iterate_for_as_while.1`.

`iterate_for_as_while.2:`

```
{1}  ∀ j:
       (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
     sult[State]],
           expr: [State → ExprResult[State, ExprData]]):
         j > 0 ⊃
          iterate_for(j, b_ex, expr, body) =
           (e2s(b_ex) ## body ## catch_continue ##
              iterate_while(j − 1, expr ## b_ex, body)
              ## e2s(expr)))
       ⊃
       (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
     sult[State]],
           expr: [State → ExprResult[State, ExprData]]):
         j + 1 > 0 ⊃
          iterate_for(j + 1, b_ex, expr, body) =
           (e2s(b_ex) ## body ## catch_continue ##
              iterate_while(j + 1 − 1, expr ## b_ex, body)
              ## e2s(expr)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Applying decompose-equality,

Case splitting on j!1 = 0,

we get 2 subgoals:

`iterate_for_as_while.2.1:`

```
{-1}  j′ = 0
{-2}  j′ ≥ 0
{-3}  j′ > 0 ⊃
       iterate_for(j′, b_ex′, expr′, body′) =
        (e2s(b_ex′) ## body′ ## catch_continue ##
           iterate_while(j′ − 1, expr′ ## b_ex′, body′)
           ## e2s(expr′))
{-4}  1 + j′ > 0
-------------------------------------------------------------------------------
 {1}  iterate_for(1 + j′, b_ex′, expr′, body′)(x′) =
       (e2s(b_ex′) ## body′ ## catch_continue ## iterate_while(j′, expr′ ## b_ex′, body′) ##
          e2s(expr′))
           (x′)
```

Replacing using formula -1,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `iterate_for_as_while.2.1`.

`iterate_for_as_while.2.2:`

{-1} $j' \geq 0$
{-2} $j' > 0 \supset$
iterate_for(j', b_ex′, expr′, body′) =
(e2s(b_ex′) ## body′ ## catch_continue ##
iterate_while($j' - 1$, expr′ ## b_ex′, body′)
e2s(expr′))
{-3} $1 + j' > 0$

{1} $j' = 0$
{2} iterate_for($1 + j'$, b_ex′, expr′, body′)(x') =
(e2s(b_ex′) ## body′ ## catch_continue ## iterate_while(j', expr′ ## b_ex′, body′) ##
e2s(expr′))
(x')

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of iterate_for,
Expanding the definition of iterate_while,
Replacing using formula -2,
Keeping (2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_for_as_while.2.2`.
`iterate_for_as_while.3:`

{1} ∀ (n: nat): $n > 0 \supset n - 1 \geq 0$
{2} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
expr: [State → ExprResult[State, ExprData]]):
$n' > 0 \supset$
iterate_for(n', b_ex, expr, body) =
(e2s(b_ex) ## body ## catch_continue ## iterate_while($n' - 1$, expr ## b_ex, body)
e2s(expr))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_for_as_while.3`.
Q.E.D.

C.108.5 IterationStatements2.for_termination_point?_TCC1

Terse proof for `for_termination_point?_TCC1`.
`for_termination_point?_TCC1:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]],
expr: [State → ExprResult[State, ExprData]], body: [State → StmtResult[State]],
s: State, n: nat, res: StmtResult[State]):
OK?(res) ∧ res = iterate_for(n, b_ex, expr, body)(s) ⊃
OK?[State](res) ∨
Break?[State](res) ∨
Continue?[State](res) ∨
Return?[State](res) ∨
Switch?[State](res) ∨ Default?[State](res) ∨ Exception?[State](res)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `for_termination_point?_TCC1`.
Q.E.D.

C.108.6 IterationStatements2.termination_point_for_as_while_TCC1

Terse proof for `termination_point_for_as_while_TCC1`.
`termination_point_for_as_while_TCC1:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          s: State, n: nat):
       n > 0 ∧ OK?((e2s(b_ex) ## body ## catch_continue)(s)) ⊃
        OK?[State]
            ((##[State]
                  (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
                (s))
         ∨
         Break?[State]
             ((##[State]
                   (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
               (s))
          ∨
          Continue?[State]
              ((##[State]
                    (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
                (s))
           ∨
           Return?[State]
               ((##[State]
                     (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
                 (s))
            ∨
            Switch?[State]
                ((##[State]
                      (##[State](e2s[State, boolean](b_ex), body), catch_continue[State]))
                  (s))
             ∨
             Default?[State]
                 ((##[State]
                       (##[State](e2s[State, boolean](b_ex), body),
                          catch_continue[State]))
                   (s))
              ∨
              Exception?[State]
                  ((##[State]
                        (##[State](e2s[State, boolean](b_ex), body),
                           catch_continue[State]))
                    (s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `termination_point_for_as_while_TCC1`.

Q.E.D.

C.108.7 IterationStatements2.termination_point_for_as_while

Terse proof for `termination_point_for_as_while`.

`termination_point_for_as_while`:

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], s: State, n: nat): $n > 0 \wedge$ OK?((e2s(b_ex) ## body ## catch_continue)(s)) ⊃ for_termination_point?(b_ex, expr, body, s)(n) = while_termination_point?(expr ## b_ex, body, state((e2s(b_ex) ## body ## catch_continue)(s))) ($n - 1$)

Repeatedly Skolemizing and flattening,

Expanding the definition of for_termination_point?,

Expanding the definition of while_termination_point?,

Rewriting using iterate_for_as_while, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `termination_point_for_as_while`.

Q.E.D.

C.108.8 IterationStatements2.min_termination_point_for_as_while_TCC1

Terse proof for `min_termination_point_for_as_while_TCC1`.

`min_termination_point_for_as_while_TCC1`:

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], s: State): (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)) ∧ OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) ⊃ nonempty?[nat](for_termination_point?(b_ex, expr, body, s))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `min_termination_point_for_as_while_TCC1`.

Q.E.D.

C.108.9 IterationStatements2.min_termination_point_for_as_while_TCC2

Terse proof for `min_termination_point_for_as_while_TCC2`.

`min_termination_point_for_as_while_TCC2:`

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `min_termination_point_for_as_while_TCC2`.

Q.E.D.

C.108.10 IterationStatements2.min_termination_point_for_as_while_TCC3

Terse proof for `min_termination_point_for_as_while_TCC3`.

`min_termination_point_for_as_while_TCC3:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          expr: [State → ExprResult[State, ExprData]], s: State):
        (¬ min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 ∧
          OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) ∧
           (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)))
         ⊃
         nonempty?[nat]
             (while_termination_point?[State]
                    (##[State, ExprData, boolean](expr, b_ex), body,
                      state[State]
                           ((##[State]
                                  (##[State](e2s[State, boolean](b_ex), body),
                                    catch_continue[State]))
                                (s))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Case splitting on n!1 = 0,

we get 2 subgoals:

`min_termination_point_for_as_while_TCC3.1:`

```
{-1}  n′ = 0
{-2}  n′ ≥ 0
{-3}  OK?[State]((e2s(b_ex′) ## body′ ## catch_continue)(s′))
{-4}  for_termination_point?(b_ex′, expr′, body′, s′)(n′)
-------------------------------------------------------------
{1}   min[nat](for_termination_point?(b_ex′, expr′, body′, s′)) = 0
{2}   nonempty?[nat]
          (while_termination_point?[State]
                 (##[State, ExprData, boolean](expr′, b_ex′), body′,
                   state[State]
                        ((##[State]
                               (##[State](e2s[State, boolean](b_ex′), body′),
                                 catch_continue[State]))
                             (s′))))
```

Rewriting using min_def, matching in *,

Expanding the definition of minimum?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `min_termination_point_for_as_while_TCC3.1`.

`min_termination_point_for_as_while_TCC3.2:`

```
{-1}  n′ ≥ 0
{-2}  OK?[State]((e2s(b_ex′) ## body′ ## catch_continue)(s′))
{-3}  for_termination_point?(b_ex′, expr′, body′, s′)(n′)
|-------
{1}   n′ = 0
{2}   min[nat](for_termination_point?(b_ex′, expr′, body′, s′)) = 0
{3}   nonempty?[nat]
        (while_termination_point?[State]
           (##[State, ExprData, boolean](expr′, b_ex′), body′,
            state[State]
              ((##[State]
                  (##[State](e2s[State, boolean](b_ex′), body′),
                   catch_continue[State]))
                 (s′))))
```

Rewriting using termination_point_for_as_while, matching in *,

Hiding formulas: 2,

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Expanding the definition of member,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `min_termination_point_for_as_while_TCC3.2`.

Q.E.D.

C.108.11 IterationStatements2.min_termination_point_for_as_while

Terse proof for `min_termination_point_for_as_while`.

`min_termination_point_for_as_while:`

```
|-------
{1}   ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
           expr: [State → ExprResult[State, ExprData]], s: State):
        (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)) ∧
         OK?[State]((e2s(b_ex) ## body ## catch_continue)(s)) ∧
          ¬ min[nat](for_termination_point?(b_ex, expr, body, s)) = 0
         ⊃
         min[nat](for_termination_point?(b_ex, expr, body, s)) =
          min [nat](while_termination_point?(expr ## b_ex, body, state [State]((e2s(b_ex) ## body ## cat
```

Skolemizing (with typepred on new Skolem constants),

Applying disjunctive simplification to flatten sequent,

Case splitting on while_termination_point?(expr!1 ## b_ex!1, body!1, state((e2s(b_ex!1) ## body!1 ## catch_continue) (s!1))) = (LAMBDA (n: nat): for_termination_point?(b_ex!1, expr!1, body!1, s!1)(n + 1)),

we get 2 subgoals:

`min_termination_point_for_as_while.1:`

```
{-1}  while_termination_point?(expr′ ## b_ex′, body′,
                                 state((e2s(b_ex′) ## body′ ## catch_continue)(s′)))
        = (λ (n: nat): for_termination_point?(b_ex′, expr′, body′, s′)(n + 1))
{-2}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, s′)(n)
{-3}  OK?[State]((e2s(b_ex′) ## body′ ## catch_continue)(s′))
-------------------------------------------------------------------------------
{1}   min[nat](for_termination_point?(b_ex′, expr′, body′, s′)) = 0
{2}   min[nat](for_termination_point?(b_ex′, expr′, body′, s′)) =
        min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## bod
```

Replacing using formula -1,
Keeping (-2 1 2) and hiding *,
Using lemma min_plus1,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: 2,
Rewriting using min_def, matching in *,
Expanding the definition of minimum?,
which is trivially true.
This completes the proof of `min_termination_point_for_as_while.1`.
`min_termination_point_for_as_while.2:`

```
{-1}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, s′)(n)
{-2}  OK?[State]((e2s(b_ex′) ## body′ ## catch_continue)(s′))
-------------------------------------------------------------------------------
{1}   while_termination_point?(expr′ ## b_ex′, body′,
                                 state((e2s(b_ex′) ## body′ ## catch_continue)(s′)))
        = (λ (n: nat): for_termination_point?(b_ex′, expr′, body′, s′)(n + 1))
{2}   min[nat](for_termination_point?(b_ex′, expr′, body′, s′)) = 0
{3}   min[nat](for_termination_point?(b_ex′, expr′, body′, s′)) =
        min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## bod
```

Hiding formulas: 3,
Applying decompose-equality,
Rewriting using termination_point_for_as_while, matching in *,
This completes the proof of `min_termination_point_for_as_while.2`.
Q.E.D.

C.108.12 IterationStatements2.nonzero_min_for_termination_point_TCC1

Terse proof for `nonzero_min_for_termination_point_TCC1`.
`nonzero_min_for_termination_point_TCC1:`

```
-------------------------------------------------------------------------------
{1}   ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
           expr: [State → ExprResult[State, ExprData]], s: State):
        (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)) ⊃
          nonempty?[nat](for_termination_point?(b_ex, expr, body, s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nonzero_min_for_termination_point_TCC1`.
Q.E.D.

C.108.13 IterationStatements2.nonzero_min_for_termination_point

Terse proof for `nonzero_min_for_termination_point`.

`nonzero_min_for_termination_point`:

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], s: State): (∃ (n: nat): for_termination_point?(b_ex, expr, body, s)(n)) ∧ ¬ min[nat](for_termination_point?(b_ex, expr, body, s)) = 0 ⊃ OK?(b_ex(s)) ∧ data(b_ex(s))

Repeatedly Skolemizing and flattening,

Rewriting using min_def, matching in *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `nonzero_min_for_termination_point`.

Q.E.D.

C.108.14 IterationStatements2.for_as_while

Terse proof for `for_as_while`.

`for_as_while`:

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], expr: [State → ExprResult[State, ExprData]], init: [State → StmtResult[State]]): for(init, b_ex, expr, body) = (init ## if_else(b_ex, body ## catch_continue ## while(expr ## b_ex, body) ## catch_break, skip))

Installing automatic rewrites from: for while comp_eval_if_ok_fstmt_single lift skip

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Expanding the definition of if_else,

Expanding the definition of while,

Expanding the definition of ##,

Case splitting on OK?(init!1(x!1)),

we get 2 subgoals:

`for_as_while.1:`

```
{-1}  OK?(init′(x′))
-------------------------------------------------------------------------------
{1}   lift(λ (state: State):
             IF ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state)(n)
               THEN (iterate_for(min[nat]
                                 (for_termination_point?(b_ex′, expr′, body′, state)),
                                 b_ex′, expr′, body′)
                     ## e2s(b_ex′)
                     ## catch_break)
                      (state)
             ELSE Hang
             ENDIF)
        (init′(x′))
      =
      lift((b_ex′ ##
            (λ (b: bool):
               IF b
                 THEN (body′ ## catch_continue ##
                         (λ (s: State):
                            IF ∃ (n: nat):
                                 while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                              THEN (iterate_while(min[nat]
                                                  (while_termination_point?(expr′
                                                                            expr′ ## b_ex′, body′)
                                                  ## e2s(expr′ ## b_ex′)
                                                  ## catch_break)
                                                   (s)
                                   ELSE Hang
                                   ENDIF))
                       ## catch_break
               ELSE skip
               ENDIF)))
          (init′(x′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

for_as_while.1.1:

```
{-1}  OK?(init′(x′))
{-2}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
{1}   (iterate_for(min[nat]
                   (for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))),
                   b_ex′, expr′, body′)
        ## e2s(b_ex′)
        ## catch_break)
         (state(init′(x′)))
      =
      (b_ex′ ##
         (λ (b: bool):
            IF b
              THEN (body′ ## catch_continue ##
                      (λ (s: State):
                         IF ∃ (n: nat):
                              while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                            THEN (iterate_while(min[nat]
                                                (while_termination_point?(expr′ ## b_ex′,
                                                                          body′,
                                                                          s)),
                                                expr′ ## b_ex′, body′)
                                   ## e2s(expr′ ## b_ex′)
                                   ## catch_break)
                                    (s)
                            ELSE Hang
                            ENDIF))
                    ## catch_break
              ELSE skip
              ENDIF))
          (state(init′(x′)))
```

Case splitting on min(for_termination_point?(b_ex!1, expr!1, body!1, state(init!1(x!1)))) = 0,

we get 2 subgoals:

`for_as_while.1.1.1:`

```
{-1}  min(for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))) = 0
{-2}  OK?(init′(x′))
{-3}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
------------------------------------------------------------------------------
{1}   (iterate_for(min[nat]
                   (for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))),
                   b_ex′, expr′, body′)
       ## e2s(b_ex′)
       ## catch_break)
        (state(init′(x′)))
      =
      (b_ex′ ##
         (λ (b: bool):
            IF b
              THEN (body′ ## catch_continue ##
                      (λ (s: State):
                         IF ∃ (n: nat):
                              while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                           THEN (iterate_while(min[nat]
                                               (while_termination_point?(expr′ ## b
                                                                         body′
                                                                         s)),
                                               expr′ ## b_ex′, body′)
                                  ## e2s(expr′ ## b_ex′)
                                  ## catch_break)
                                   (s)
                           ELSE Hang
                           ENDIF))
                    ## catch_break
              ELSE skip
              ENDIF))
          (state(init′(x′)))
```

Replacing using formula -1,

Expanding the definition of iterate_for,

Rewriting using min_def, matching in *,

Hiding formulas: -3,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.1`.

for_as_while.1.1.2:

```
{-1}  OK?(init'(x'))
{-2}  ∃ (n: nat): for_termination_point?(b_ex', expr', body', state(init'(x')))(n)
------------------------------------------------------------------------------------
{1}   min(for_termination_point?(b_ex', expr', body', state(init'(x')))) = 0
{2}   (iterate_for(min[nat]
                   (for_termination_point?(b_ex', expr', body', state(init'(x')))),
                   b_ex', expr', body')
        ## e2s(b_ex')
        ## catch_break)
         (state(init'(x')))
       =
       (b_ex' ##
          (λ (b: bool):
             IF b
               THEN (body' ## catch_continue ##
                       (λ (s: State):
                          IF ∃ (n: nat):
                               while_termination_point?(expr' ## b_ex', body', s)(n)
                             THEN (iterate_while(min[nat]
                                                 (while_termination_point?(expr' ## b_ex',
                                                                           body',
                                                                           s)),
                                                 expr' ## b_ex', body')
                                     ## e2s(expr' ## b_ex')
                                     ## catch_break)
                                      (s)
                             ELSE Hang
                             ENDIF))
                     ## catch_break
             ELSE skip
             ENDIF))
          (state(init'(x')))
```

Using lemma nonzero_min_for_termination_point,

Rewriting using iterate_for_as_while, matching in *,

Rewriting using min_termination_point_for_as_while, matching in *,

we get 2 subgoals:

`for_as_while.1.1.2.1:`

```
{-1}  (∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n))
       ∧
      ¬ 1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) #
        = 0
      ⊃ OK?(b_ex′(state(init′(x′)))) ∧ data(b_ex′(state(init′(x′))))
{-2}  OK?(init′(x′))
{-3}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
------------------------------------------------------------------------------
{1}   1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## b
       = 0
{2}   ((e2s(b_ex′) ## body′ ## catch_continue ##
          iterate_while(1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State
                                expr′ ## b_ex′, body′)
          ## e2s(expr′))
        ## e2s(b_ex′)
        ## catch_break)
         (state(init′(x′)))
       =
       (b_ex′ ##
          (λ (b: bool):
             IF b
               THEN (body′ ## catch_continue ##
                       (λ (s: State):
                          IF ∃ (n: nat):
                                while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                            THEN (iterate_while(min[nat]
                                                  (while_termination_point?(expr′ ## b
                                                                                  body′
                                                                                  s)),
                                                       expr′ ## b_ex′, body′)
                                   ## e2s(expr′ ## b_ex′)
                                   ## catch_break)
                                    (s)
                            ELSE Hang
                            ENDIF))
                    ## catch_break
             ELSE skip
             ENDIF))
          (state(init′(x′)))
```

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of lift,

Rewriting using comp_simple_stmt_simple_expr, matching in *,

Case splitting on OK?((body!1 ## catch_continue)(state((init!1 ## b_ex!1)(x!1)))),

we get 2 subgoals:

for_as_while.1.1.2.1.1:

```
{-1}  OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{-2}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
{-3}  OK?((init′ ## b_ex′)(x′))
{-4}  data((init′ ## b_ex′)(x′))
{-5}  OK?(init′(x′))
------------------------------------------------------------------------------
{1}   1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## body′ ## c
       = 0
{2}   ((e2s(b_ex′) ## body′ ## catch_continue ##
          iterate_while(min[nat]
                        (while_termination_point?(expr′ ## b_ex′, body′,
                                                  state[State]
                                                       ((e2s(b_ex′) ## body′
                                                            ##
                                                            catch_continue)
                                                          (state(init′(x′))))))),
                        expr′ ## b_ex′, body′)
        ## e2s(expr′))
       ## e2s(b_ex′)
       ## catch_break)
        (state(init′(x′)))
      =
      catch_break(CASES (body′ ## catch_continue)(state((init′ ## b_ex′)(x′))) OF
                    OK(state_1):
                      IF ∃ (n: nat):
                           while_termination_point?(expr′ ## b_ex′, body′, state_1)(n)
                      THEN (iterate_while(min[nat]
                                          (while_termination_point?(expr′ ## b_ex′,
                                                                    body′,
                                                                    state_1)),
                                          expr′ ## b_ex′, body′)
                                ## e2s(expr′ ## b_ex′)
                                ## catch_break)
                                 (state_1)
                      ELSE Hang
                      ENDIF
                    ELSE (body′ ## catch_continue)(state((init′ ## b_ex′)(x′)))
                    ENDCASES)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`for_as_while.1.1.2.1.1.1:`

Keeping (-1 -3 -5 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.1`.

`for_as_while.1.1.2.1.1.2:`

```
{-1}  OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{-2}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
{-3}  OK?((init′ ## b_ex′)(x′))
{-4}  data((init′ ## b_ex′)(x′))
{-5}  OK?(init′(x′))
-------------------------------------------------------------------------------
{1}   1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## body′ ##
        = 0
{2}   ∃ (n: nat):
        while_termination_point?(expr′ ## b_ex′, body′,
                                 state((body′ ## catch_continue)
                                       (state((init′ ## b_ex′)(x′)))))
                                (n)
{3}   ((e2s(b_ex′) ## body′ ## catch_continue ##
          iterate_while(min[nat]
                        (while_termination_point?(expr′ ## b_ex′, body′,
                                                  state[State]
                                                       ((e2s(b_ex′) ## body′
                                                            ##
                                                            catch_continue)
                                                          (state(init′(x′))))))),
                        expr′ ## b_ex′, body′)
          ## e2s(expr′))
        ## e2s(b_ex′)
        ## catch_break)
         (state(init′(x′)))
      = catch_break(Hang)
```

Hiding formulas: 3,
Repeatedly Skolemizing and flattening,
Case splitting on n!1 = 0,
we get 2 subgoals:
`for_as_while.1.1.2.1.1.2.1:`

```
{-1}  n′ = 0
{-2}  n′ ≥ 0
{-3}  OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{-4}  for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n′)
{-5}  OK?((init′ ## b_ex′)(x′))
{-6}  data((init′ ## b_ex′)(x′))
{-7}  OK?(init′(x′))
-------------------------------------------------------------------------------
{1}   1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## body′ ##
        = 0
{2}   ∃ (n: nat):
        while_termination_point?(expr′ ## b_ex′, body′,
                                 state((body′ ## catch_continue)
                                       (state((init′ ## b_ex′)(x′)))))
                                (n)
```

Replacing using formula -1,
Expanding the definition of for_termination_point?,
Expanding the definition of iterate_for,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `for_as_while.1.1.2.1.1.2.1`.

`for_as_while.1.1.2.1.1.2.2:`

{-1} $n' \geq 0$
{-2} OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{-3} for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n′)
{-4} OK?((init′ ## b_ex′)(x′))
{-5} data((init′ ## b_ex′)(x′))
{-6} OK?(init′(x′))

{1} $n' = 0$
{2} $1 + \min\ [\text{nat}](\text{while\_termination\_point?}(\text{expr}' \#\#\ \text{b\_ex}', \text{body}', \text{state}\ [\text{State}](\text{e2s}(\text{b\_ex}') \#\#\ \text{b}$
$= 0$
{3} $\exists\ (n\colon \text{nat})\colon$
while_termination_point?(expr′ ## b_ex′, body′,
state((body′ ## catch_continue)
(state((init′ ## b_ex′)(x′)))))
(n)

Instantiating the top quantifier in 3 with the terms: $n' - 1$,
we get 2 subgoals:

`for_as_while.1.1.2.1.1.2.2.1:`

{-1} $n' \geq 0$
{-2} OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{-3} for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n′)
{-4} OK?((init′ ## b_ex′)(x′))
{-5} data((init′ ## b_ex′)(x′))
{-6} OK?(init′(x′))

{1} $n' = 0$
{2} $1 + \min\ [\text{nat}](\text{while\_termination\_point?}(\text{expr}' \#\#\ \text{b\_ex}', \text{body}', \text{state}\ [\text{State}](\text{e2s}(\text{b\_ex}') \#\#\ \text{b}$
$= 0$
{3} while_termination_point?(expr′ ## b_ex′, body′,
state((body′ ## catch_continue)
(state((init′ ## b_ex′)(x′)))))
$(n' - 1)$

Rewriting using termination_point_for_as_while, matching in *,
we get 2 subgoals:

`for_as_while.1.1.2.1.1.2.2.1.1:`

{-1} $n' \geq 0$
{-2} OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{-3} while_termination_point?(expr′ ## b_ex′, body′,
state((e2s(b_ex′) ## body′ ## catch_continue)
(state(init′(x′)))))
$(n' - 1)$
{-4} OK?((init′ ## b_ex′)(x′))
{-5} data((init′ ## b_ex′)(x′))
{-6} OK?(init′(x′))

{1} $n' = 0$
{2} $1 + \min\ [\text{nat}](\text{while\_termination\_point?}(\text{expr}' \#\#\ \text{b\_ex}', \text{body}', \text{state}\ [\text{State}](\text{e2s}(\text{b\_ex}') \#\#\ \text{b}$
$= 0$
{3} while_termination_point?(expr′ ## b_ex′, body′,
state((body′ ## catch_continue)
(state((init′ ## b_ex′)(x′)))))
$(n' - 1)$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.2.2.1.1`.

`for_as_while.1.1.2.1.1.2.2.1.2`:

$$
\begin{array}{ll}
\{\text{-1}\} & n' \geq 0 \\
\{\text{-2}\} & \mathrm{OK?}((\mathrm{body}' \ \#\# \ \mathrm{catch\_continue})(\mathrm{state}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x')))) \\
\{\text{-3}\} & \mathrm{for\_termination\_point?}(\mathrm{b\_ex}', \ \mathrm{expr}', \ \mathrm{body}', \ \mathrm{state}(\mathrm{init}'(x')))(n') \\
\{\text{-4}\} & \mathrm{OK?}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x')) \\
\{\text{-5}\} & \mathrm{data}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x')) \\
\{\text{-6}\} & \mathrm{OK?}(\mathrm{init}'(x')) \\
\hline
\{1\} & \mathrm{OK?}((\mathrm{e2s}(\mathrm{b\_ex}') \ \#\# \ \mathrm{body}' \ \#\# \ \mathrm{catch\_continue})(\mathrm{state}(\mathrm{init}'(x')))) \\
\{2\} & n' = 0 \\
\{3\} & 1 + \min\,[\mathrm{nat}](\mathrm{while\_termination\_point?}(\mathrm{expr}' \ \#\# \ \mathrm{b\_ex}', \ \mathrm{body}', \ \mathrm{state}\,[\mathrm{State}]((\mathrm{e2s}(\mathrm{b\_ex}') \ \#\# \ \mathrm{body}' \ \#\# \ \\
 & \quad = 0 \\
\{4\} & \mathrm{while\_termination\_point?}(\mathrm{expr}' \ \#\# \ \mathrm{b\_ex}', \ \mathrm{body}', \\
 & \qquad \mathrm{state}((\mathrm{body}' \ \#\# \ \mathrm{catch\_continue}) \\
 & \qquad\quad (\mathrm{state}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x'))))) \\
 & \qquad (n' - 1)
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.2.2.1.2`.

`for_as_while.1.1.2.1.1.2.2.2`:

$$
\begin{array}{ll}
\{\text{-1}\} & n' \geq 0 \\
\{\text{-2}\} & \mathrm{OK?}((\mathrm{body}' \ \#\# \ \mathrm{catch\_continue})(\mathrm{state}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x')))) \\
\{\text{-3}\} & \mathrm{for\_termination\_point?}(\mathrm{b\_ex}', \ \mathrm{expr}', \ \mathrm{body}', \ \mathrm{state}(\mathrm{init}'(x')))(n') \\
\{\text{-4}\} & \mathrm{OK?}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x')) \\
\{\text{-5}\} & \mathrm{data}((\mathrm{init}' \ \#\# \ \mathrm{b\_ex}')(x')) \\
\{\text{-6}\} & \mathrm{OK?}(\mathrm{init}'(x')) \\
\hline
\{1\} & n' - 1 \geq 0 \\
\{2\} & n' = 0 \\
\{3\} & 1 + \min\,[\mathrm{nat}](\mathrm{while\_termination\_point?}(\mathrm{expr}' \ \#\# \ \mathrm{b\_ex}', \ \mathrm{body}', \ \mathrm{state}\,[\mathrm{State}]((\mathrm{e2s}(\mathrm{b\_ex}') \ \#\# \ \mathrm{body}' \ \#\# \ \\
 & \quad = 0
\end{array}
$$

Keeping (-1 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.1.2.2.2`.

`for_as_while.1.1.2.1.2:`

```
{-1}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
{-2}  OK?((init′ ## b_ex′)(x′))
{-3}  data((init′ ## b_ex′)(x′))
{-4}  OK?(init′(x′))
------------------------------------------------------------------------------
{1}   OK?((body′ ## catch_continue)(state((init′ ## b_ex′)(x′))))
{2}   1 + min [nat](while_termination_point?(expr′ ## b_ex′, body′, state [State]((e2s(b_ex′) ## b
       = 0
{3}   ((e2s(b_ex′) ## body′ ## catch_continue ##
          iterate_while(min[nat]
                                (while_termination_point?(expr′ ## b_ex′, body′,
                                                          state[State]
                                                              ((e2s(b_ex′) ## body′
                                                                  ##
                                                                  catch_continue)
                                                                 (state(init′(x′))))))),
                                expr′ ## b_ex′, body′)
            ## e2s(expr′))
        ## e2s(b_ex′)
        ## catch_break)
         (state(init′(x′)))
      =
      catch_break(CASES (body′ ## catch_continue)(state((init′ ## b_ex′)(x′))) OF
                        OK(state_1):
                          IF ∃ (n: nat):
                                while_termination_point?(expr′ ## b_ex′, body′, state_1)(n)
                            THEN (iterate_while(min[nat]
                                                    (while_termination_point?(expr′ ## b_ex′,
                                                                              body′,
                                                                              state_1)),
                                                    expr′ ## b_ex′, body′)
                                  ## e2s(expr′ ## b_ex′)
                                  ## catch_break)
                                   (state_1)
                            ELSE Hang
                            ENDIF
                        ELSE (body′ ## catch_continue)(state((init′ ## b_ex′)(x′)))
                        ENDCASES)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.1.2`.

`for_as_while.1.1.2.2:`

```
{-1}  (∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n))
        ⊃ OK?(b_ex′(state(init′(x′)))) ∧ data(b_ex′(state(init′(x′))))
{-2}  OK?(init′(x′))
{-3}  ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
------------------------------------------------------------------------------------
{1}   OK?[State]((e2s(b_ex′) ## body′ ## catch_continue)(state(init′(x′))))
{2}   min(for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))) = 0
{3}   ((e2s(b_ex′) ## body′ ## catch_continue ##
          iterate_while(min [nat](for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))) − 1,
                        expr′ ## b_ex′, body′)
          ## e2s(expr′))
        ## e2s(b_ex′)
        ## catch_break)
         (state(init′(x′)))
       =
       (b_ex′ ##
          (λ (b: bool):
             IF b
               THEN (body′ ## catch_continue ##
                       (λ (s: State):
                          IF ∃ (n: nat):
                               while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                             THEN (iterate_while(min[nat]
                                                 (while_termination_point?(expr′ ## b_ex′,
                                                                           body′,
                                                                           s)),
                                                 expr′ ## b_ex′, body′)
                                     ## e2s(expr′ ## b_ex′)
                                     ## catch_break)
                                      (s)
                             ELSE Hang
                             ENDIF))
                     ## catch_break
               ELSE skip
               ENDIF))
            (state(init′(x′)))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.1.2.2`.

`for_as_while.1.2:`

```
{-1}  OK?(init′(x′))
{1}   ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state(init′(x′)))(n)
{2}   Hang?((b_ex′ ##
               (λ (b: bool):
                  IF b
                    THEN (body′ ## catch_continue ##
                            (λ (s: State):
                               IF ∃ (n: nat):
                                    while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                                 THEN (iterate_while(min[nat]
                                                     (while_termination_point?(expr′
                                                                               expr′ ## b_ex′, body′)
                                                     ## e2s(expr′ ## b_ex′)
                                                     ## catch_break)
                                                     (s)
                                       ELSE Hang
                                       ENDIF))
                          ## catch_break
                    ELSE skip
                    ENDIF))
               (state(init′(x′))))
```

Copying formula number: 1

Copying formula number: 1

Instantiating the top quantifier in 1 with the terms: 0,

Instantiating the top quantifier in 2 with the terms: 1,

Expanding the definition of for_termination_point?,

Expanding the definition of iterate_for,

Expanding the definition of iterate_for,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ##,

Expanding the definition of catch_break,

Expanding the definition of lift,

Expanding the definition of e2s,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of catch_continue,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on OK?(body!1(state(b_ex!1(state(init!1(x!1)))))) OR Continue?(body!1(state(b_ex!1(state(init!1(

we get 2 subgoals:

```
for_as_while.1.2.1:
  {-1}  OK?(body′(state(b_ex′(state(init′(x′)))))) ∨
        Continue?(body′(state(b_ex′(state(init′(x′))))))
  {-2}  OK?(OK(state(init′(x′))))
  {-3}  OK?(b_ex′(state(init′(x′))))
  {-4}  data(b_ex′(state(init′(x′))))
  {-5}  OK?(((((λ (s: State):
                  CASES b_ex′(s) OF
                    OK(state, data): OK(state),
                    Fatal: Fatal,
                    Hang: Hang,
                    Exception(ex_1, state): Exception(ex_1, state)
                  ENDCASES)
                ## body′
                ##
                (λ (res: StmtResult[State]):
                    CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
              ##
              (λ (s: State):
                  CASES expr′(s) OF
                    OK(state, data): OK(state),
                    Fatal: Fatal,
                    Hang: Hang,
                    Exception(ex_1, state): Exception(ex_1, state)
                  ENDCASES))
            ## (λ (s): OK(s)))
           (state(init′(x′))))
  {-6}  OK?(b_ex′(state(((((λ (s: State):
                                CASES b_ex′(s) OF
                                  OK(state, data): OK(state),
                                  Fatal: Fatal,
                                  Hang: Hang,
                                  Exception(ex_1, state): Exception(ex_1, state)
                                ENDCASES)
                              ## body′
                              ##
                              (λ (res: StmtResult[State]):
                                  CASES res OF Continue(state): OK(state) ELSE res END-
      CASES))
                            ##
                            (λ (s: State):
                                CASES expr′(s) OF
                                  OK(state, data): OK(state),
                                  Fatal: Fatal,
                                  Hang: Hang,
                                  Exception(ex_1, state): Exception(ex_1, state)
                                ENDCASES))
                          ## (λ (s): OK(s)))
                         (state(init′(x′))))))
  {-7}  data(b_ex′(state(((((λ (s: State):
                                CASES b_ex′(s) OF
                                  OK(state, data): OK(state),
                                  Fatal: Fatal,
                                  Hang: Hang,
                                  Exception(ex_1, state): Exception(ex_1, state)
                                ENDCASES)
                              ## body′                                    1311
                              ##
                              (λ (res: StmtResult[State]):
                                  CASES res OF Continue(state): OK(state) ELSE res END-
      CASES))
                            ##
                            (λ (s: State):
                                CASES expr′(s) OF
```

Case splitting on EXISTS (n: nat): while_termination_point?(expr!1 ## b_ex!1, body!1, state(body!1 (state(b_ex!1(state(init!1(x!1))))))) (n),

we get 3 subgoals:

```
for_as_while.1.2.1.1:
```

{-1} ∃ (n: nat):
  while_termination_point?(expr′ ## b_ex′, body′,
    state(body′(state(b_ex′(state(init′(x')))))))
    (n)
{-2} OK?(body′(state(b_ex′(state(init′(x')))))) ∨
  Continue?(body′(state(b_ex′(state(init′(x'))))))
{-3} OK?(OK(state(init′(x'))))
{-4} OK?(b_ex′(state(init′(x'))))
{-5} data(b_ex′(state(init′(x'))))
{-6} OK?(((((λ (s: State):
    CASES b_ex′(s) OF
      OK(state, data): OK(state),
      Fatal: Fatal,
      Hang: Hang,
      Exception(ex_1, state): Exception(ex_1, state)
    ENDCASES)
  ## body′
  ##
  (λ (res: StmtResult[State]):
    CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
 ##
 (λ (s: State):
    CASES expr′(s) OF
      OK(state, data): OK(state),
      Fatal: Fatal,
      Hang: Hang,
      Exception(ex_1, state): Exception(ex_1, state)
    ENDCASES))
 ## (λ (s): OK(s)))
 (state(init′(x'))))
{-7} OK?(b_ex′(state(((((λ (s: State):
    CASES b_ex′(s) OF
      OK(state, data): OK(state),
      Fatal: Fatal,
      Hang: Hang,
      Exception(ex_1, state): Exception(ex_1, state)
    ENDCASES)
  ## body′
  ##
  (λ (res: StmtResult[State]):
    CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
 ##
 (λ (s: State):
    CASES expr′(s) OF
      OK(state, data): OK(state),
      Fatal: Fatal,
      Hang: Hang,
      Exception(ex_1, state): Exception(ex_1, state)
    ENDCASES))
 ## (λ (s): OK(s)))
 (state(init′(x'))))))
{-8} data(b_ex′(state(((((λ (s: State):
    CASES b_ex′(s) OF
      OK(state, data): OK(state),
      Fatal: Fatal,
      Hang: Hang,
      Exception(ex_1, state): Exception(ex_1, state)
    ENDCASES)
  ## body′
  ##
  (λ (res: StmtResult[State]):
    CASES res OF Continue(state): OK(state) ELSE res END-

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $n' + 1$,

Rewriting using termination_point_for_as_while, matching in *,

we get 2 subgoals:

for_as_while.1.2.1.1.1:

{-1} $n' \geq 0$

{-2} while_termination_point?(expr′ ## b_ex′, body′, state(body′(state(b_ex′(state(init′(x′)))))))(n′)

{-3} OK?(body′(state(b_ex′(state(init′(x′)))))) ∨ Continue?(body′(state(b_ex′(state(init′(x′))))))

{-4} OK?(OK(state(init′(x′))))

{-5} OK?(b_ex′(state(init′(x′))))

{-6} data(b_ex′(state(init′(x′))))

{-7} OK?(((((λ (s: State): CASES b_ex′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES) ## body′ ## (λ (res: StmtResult[State]): CASES res OF Continue(state): OK(state) ELSE res ENDCASES)) ## (λ (s: State): CASES expr′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES)) ## (λ (s): OK(s))) (state(init′(x′))))

{-8} OK?(b_ex′(state(((((λ (s: State): CASES b_ex′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES) ## body′ ## (λ (res: StmtResult[State]): CASES res OF Continue(state): OK(state) ELSE res ENDCASES)) ## (λ (s: State): CASES expr′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES)) ## (λ (s): OK(s))) (state(init′(x′))))))

{-9} data(b_ex′(state(((((λ (s: State): CASES b_ex′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES) ## body′ ## (λ (res: StmtResult[State]): CASES res OF Continue(state): OK(state) ELSE res END-

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.2.1.1.1`.

for_as_while.1.2.1.1.2:

{-1} $n' \geq 0$
{-2} while_termination_point?(expr′ ## b_ex′, body′, state(body′(state(b_ex′(state(init′(x′)))))))(n′)
{-3} OK?(body′(state(b_ex′(state(init′(x′)))))) ∨ Continue?(body′(state(b_ex′(state(init′(x′))))))
{-4} OK?(OK(state(init′(x′))))
{-5} OK?(b_ex′(state(init′(x′))))
{-6} data(b_ex′(state(init′(x′))))
{-7} OK?(((((λ (s: State): CASES b_ex′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES) ## body′ ## (λ (res: StmtResult[State]): CASES res OF Continue(state): OK(state) ELSE res ENDCASES)) ## (λ (s: State): CASES expr′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES)) ## (λ (s): OK(s))) (state(init′(x′))))
{-8} OK?(b_ex′(state(((((λ (s: State): CASES b_ex′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES) ## body′ ## (λ (res: StmtResult[State]): CASES res OF Continue(state): OK(state) ELSE res ENDCASES)) ## (λ (s: State): CASES expr′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES)) ## (λ (s): OK(s))) (state(init′(x′))))))
{-9} data(b_ex′(state(((((λ (s: State): CASES b_ex′(s) OF OK(state, data): OK(state), Fatal: Fatal, Hang: Hang, Exception(ex_1, state): Exception(ex_1, state) ENDCASES) ## body′ ## (λ (res: StmtResult[State]): CASES res OF Continue(state): OK(state) ELSE res END-

Hiding formulas: (-2 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.2.1.1.2`.

for_as_while.1.2.1.2:

{-1} OK?(body′(state(b_ex′(state(init′(x′)))))) ∨
Continue?(body′(state(b_ex′(state(init′(x′))))))
{-2} OK?(OK(state(init′(x′))))
{-3} OK?(b_ex′(state(init′(x′))))
{-4} data(b_ex′(state(init′(x′))))
{-5} OK?(((((λ (s: State):
cases b_ex′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases)
body′
##
(λ (res: StmtResult[State]):
cases res of Continue(state): OK(state) else res endcases))
##
(λ (s: State):
cases expr′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases))
(λ (s): OK(s)))
(state(init′(x′))))
{-6} OK?(b_ex′(state(((((λ (s: State):
cases b_ex′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases)
body′
##
(λ (res: StmtResult[State]):
cases res of Continue(state): OK(state) else res endcases))
##
(λ (s: State):
cases expr′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases))
(λ (s): OK(s)))
(state(init′(x′))))))
{-7} data(b_ex′(state(((((λ (s: State):
cases b_ex′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases)
body′
##
(λ (res: StmtResult[State]):
cases res of Continue(state): OK(state) else res endcases))
##
(λ (s: State):
cases expr′(s) of

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `for_as_while.1.2.1.2`.

for_as_while.1.2.1.3:

{-1} OK?(body′(state(b_ex′(state(init′(x′)))))) ∨
Continue?(body′(state(b_ex′(state(init′(x′))))))
{-2} OK?(OK(state(init′(x′))))
{-3} OK?(b_ex′(state(init′(x′))))
{-4} data(b_ex′(state(init′(x′))))
{-5} OK?(((((λ (s: State):
cases b_ex′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases)
body′
##
(λ (res: StmtResult[State]):
cases res of Continue(state): OK(state) else res endcases))
##
(λ (s: State):
cases expr′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases))
(λ (s): OK(s)))
(state(init′(x′))))
{-6} OK?(b_ex′(state(((((λ (s: State):
cases b_ex′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases)
body′
##
(λ (res: StmtResult[State]):
cases res of Continue(state): OK(state) else res endcases))
##
(λ (s: State):
cases expr′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases))
(λ (s): OK(s)))
(state(init′(x′))))))
{-7} data(b_ex′(state(((((λ (s: State):
cases b_ex′(s) of
OK(state, data): OK(state),
Fatal: Fatal,
Hang: Hang,
Exception(ex_1, state): Exception(ex_1, state)
endcases)
body′
##
(λ (res: StmtResult[State]):
cases res of Continue(state): OK(state) else res endcases))
##
(λ (s: State):
cases expr′(s) of

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `for_as_while.1.2.1.3`.

```
for_as_while.1.2.2:
```

```
{-1}  OK?(OK(state(init′(x′))))
{-2}  OK?(b_ex′(state(init′(x′))))
{-3}  data(b_ex′(state(init′(x′))))
{-4}  OK?(((((λ (s: State):
                 CASES b_ex′(s) OF
                   OK(state, data): OK(state),
                   Fatal: Fatal,
                   Hang: Hang,
                   Exception(ex_1, state): Exception(ex_1, state)
                 ENDCASES)
               ## body′
               ##
               (λ (res: StmtResult[State]):
                  CASES res OF Continue(state): OK(state) ELSE res ENDCASES))
             ##
             (λ (s: State):
                CASES expr′(s) OF
                  OK(state, data): OK(state),
                  Fatal: Fatal,
                  Hang: Hang,
                  Exception(ex_1, state): Exception(ex_1, state)
                  ENDCASES))
           ## (λ (s): OK(s)))
            (state(init′(x′))))
{-5}  OK?(b_ex′(state(((((λ (s: State):
                              CASES b_ex′(s) OF
                                OK(state, data): OK(state),
                                Fatal: Fatal,
                                Hang: Hang,
                                Exception(ex_1, state): Exception(ex_1, state)
                                ENDCASES)
                            ## body′
                            ##
                            (λ (res: StmtResult[State]):
                               CASES res OF Continue(state): OK(state) ELSE res END-
      CASES))
                           ##
                           (λ (s: State):
                              CASES expr′(s) OF
                                OK(state, data): OK(state),
                                Fatal: Fatal,
                                Hang: Hang,
                                Exception(ex_1, state): Exception(ex_1, state)
                                ENDCASES))
                         ## (λ (s): OK(s)))
                          (state(init′(x′))))))
{-6}  data(b_ex′(state(((((λ (s: State):
                              CASES b_ex′(s) OF
                                OK(state, data): OK(state),
                                Fatal: Fatal,
                                Hang: Hang,
                                Exception(ex_1, state): Exception(ex_1, state)
                                ENDCASES)
                            ## body′
                            ##
                            (λ (res: StmtResult[State]):
                               CASES res OF Continue(state): OK(state) ELSE res END-
      CASES))
                           ##
                           (λ (s: State):
                              CASES expr′(s) OF
                                OK(state, data): OK(state),
                                Fatal: Fatal,
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `for_as_while.1.2.2`.

`for_as_while.2`:

```
{1}   OK?(init′(x′))
{2}   lift(λ (state: State):
             IF ∃ (n: nat): for_termination_point?(b_ex′, expr′, body′, state)(n)
               THEN (iterate_for(min⌈nat⌉
                                 (for_termination_point?(b_ex′, expr′, body′, state)),
                                 b_ex′, expr′, body′)
                     ## e2s(b_ex′)
                     ## catch_break)
                      (state)
             ELSE Hang
             ENDIF)
        (init′(x′))
      =
      lift((b_ex′ ##
            (λ (b: bool):
               IF b
                 THEN (body′ ## catch_continue ##
                       (λ (s: State):
                          IF ∃ (n: nat):
                             while_termination_point?(expr′ ## b_ex′, body′, s)(n)
                            THEN (iterate_while(min⌈nat⌉
                                                (while_termination_point?(expr′
                                                                          expr′ ## b_ex′, body′)
                                 ## e2s(expr′ ## b_ex′)
                                 ## catch_break)
                                  (s)
                          ELSE Hang
                          ENDIF))
                       ## catch_break
               ELSE skip
               ENDIF)))
          (init′(x′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `for_as_while.2`.

Q.E.D.

C.108.15 IterationStatements2.for_unroll

Terse proof for `for_unroll`.

`for_unroll`:

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          expr: [State → ExprResult[State, ExprData]], init: [State → StmtRe-
      sult[State]],
          max: nat):
        for(max, init, b_ex, expr, body) =
         (init ##
            if_else(b_ex, body ## catch_continue ## while(max, expr ## b_ex, body) ## catch_break,
                    skip))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of for,
Expanding the definition of while,
Rewriting using for_as_while, matching in *,
This completes the proof of `for_unroll`.
Q.E.D.

C.109 Proofs for IterationStatements3 (statements.pvs)

C.109.1 IterationStatements3.do_while_inv_unroll

Terse proof for `do_while_inv_unroll`.

`do_while_inv_unroll`:

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
      sult[State]],
          variant: [State → [State → Ord]]):
        do_while(body, b_ex)(P, Q, R, variant, <) =
          (body ## catch_continue ## while(b_ex, body)(P, Q, R, vari-
      ant, <) ## catch_break)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of do_while,
Expanding the definition of while,
Rewriting using do_as_while, matching in *,
This completes the proof of `do_while_inv_unroll`.
Q.E.D.

C.110 Proofs for IterationStatements4 (statements.pvs)

C.110.1 IterationStatements4.for_inv_unroll

Terse proof for `for_inv_unroll`.

`for_inv_unroll:`

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
      sult[State]],
          expr: [State → ExprResult[State, ExprData]], init: [State → StmtRe-
      sult[State]],
          variant: [State → [State → Ord]]):
        for(init, b_ex, expr, body)(P, Q, R, variant, <) =
         (init ##
            if_else(b_ex,
                    body ## catch_continue ## while(expr ## b_ex, body)(P, Q, R, vari-
      ant, <) ##
                     catch_break,
                    skip))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of for,
Expanding the definition of while,
Rewriting using for_as_while, matching in *,
This completes the proof of `for_inv_unroll`.
Q.E.D.

C.111 Proofs for JumpStatements1 (statements.pvs)

This theory contains no provable formal statements.

C.112 Proofs for JumpStatements2 (statements.pvs)

This theory contains no provable formal statements.

C.113 Proofs for LabeledStatements (statements.pvs)

This theory contains no provable formal statements.

C.114 Proofs for Linear_Memory (linear_memory.pvs)

C.114.1 Linear_Memory.xlat_idx_TCC1

Terse proof for `xlat_idx_TCC1`.
`xlat_idx_TCC1:`

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G): offset(addr) ≥ 0
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_idx_TCC1`.
Q.E.D.

C.114.2 Linear_Memory.xlat_idx_TCC2

Terse proof for `xlat_idx_TCC2`.
`xlat_idx_TCC2:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
bus_width − (lvl + 1) × bits_per_level ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
Using lemma bus_width,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_TCC2`.
Q.E.D.

C.114.3 Linear_Memory.xlat_idx_pe_aligned_TCC1

Terse proof for `xlat_idx_pe_aligned_TCC1`.
`xlat_idx_pe_aligned_TCC1:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G): offset(base) ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_idx_pe_aligned_TCC1`.
Q.E.D.

C.114.4 Linear_Memory.xlat_idx_pe_aligned_TCC2

Terse proof for `xlat_idx_pe_aligned_TCC2`.
`xlat_idx_pe_aligned_TCC2:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
aligned?(bits_per_level + pe_size)(offset(base)) ⊃
offset(xlat_idx(lvl, base, addr)) ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_idx_pe_aligned_TCC2`.
Q.E.D.

C.114.5 Linear_Memory.xlat_idx_pe_aligned

Terse proof for `xlat_idx_pe_aligned`.
`xlat_idx_pe_aligned:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
aligned?(bits_per_level + pe_size)(offset(base)) ⊃
aligned?(pe_size)(offset(xlat_idx(lvl, base, addr)))

Repeatedly Skolemizing and flattening,
Expanding the definition of xlat_idx,
Installing automatic rewrites from: (bus_width! bits_per_level! max_level!)
Case splitting on -1 * (bits_per_level * lvl!1) - bits_per_level + bus_width >= 0,
we get 2 subgoals:

`xlat_idx_pe_aligned.1:`

{-1}	$-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width} \geq 0$
{-2}	$\text{lvl}' < \text{max\_level}$
{-3}	Mem?(base′‘type_of)
{-4}	$0 \leq$ base′‘offset
{-5}	base′‘offset $<$ max_linear_offset
{-6}	Mem?(addr′‘type_of)
{-7}	$0 \leq$ addr′‘offset
{-8}	addr′‘offset $<$ max_linear_offset
{-9}	aligned?(bits_per_level + pe_size)(offset(base′))
{1}	aligned?(pe_size) (offset (base′ + shift_bits_left(cut_bits(offset(addr′), $-1 \times$ bits_per_level $\times$ lvl′ $-$ bits_per_lev

Using lemma aligned_shift_bits_ok,

we get 2 subgoals:

`xlat_idx_pe_aligned.1.1:`

{-1}	pe_size $\leq$ pe_size $\supset$ aligned?(pe_size) (shift_bits_left(cut_bits(offset(addr′), bus_width $-$ (lvl′ + 1) $\times$ bits_per_level, bits_per_level), pe_size))
{-2}	$-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width} \geq 0$
{-3}	$\text{lvl}' < \text{max\_level}$
{-4}	Mem?(base′‘type_of)
{-5}	$0 \leq$ base′‘offset
{-6}	base′‘offset $<$ max_linear_offset
{-7}	Mem?(addr′‘type_of)
{-8}	$0 \leq$ addr′‘offset
{-9}	addr′‘offset $<$ max_linear_offset
{-10}	aligned?(bits_per_level + pe_size)(offset(base′))
{1}	aligned?(pe_size) (offset (base′ + shift_bits_left(cut_bits(offset(addr′), $-1 \times$ bits_per_level $\times$ lvl′ $-$ bits_per_le

Expanding the definition of +,

Using lemma aligned_plus,

we get 4 subgoals:

`xlat_idx_pe_aligned.1.1.1:`

```
{-1}  aligned?(min(bits_per_level + pe_size, pe_size))
                (offset(base′) + shift_bits_left(cut_bits(offset(addr′), bus_width − (lvl′ + 1) × bits_per_level, bits_pe
{-2}  pe_size ≤ pe_size ⊃
       aligned?(pe_size)
                (shift_bits_left(cut_bits(offset(addr′),
                                          bus_width − (lvl′ + 1) × bits_per_level,
                                          bits_per_level),
                                 pe_size))
{-3}  −1 × (bits_per_level × lvl′) − bits_per_level + bus_width ≥ 0
{-4}  lvl′ < max_level
{-5}  Mem?(base′`type_of)
{-6}  0 ≤ base′`offset
{-7}  base′`offset < max_linear_offset
{-8}  Mem?(addr′`type_of)
{-9}  0 ≤ addr′`offset
{-10} addr′`offset < max_linear_offset
{-11} aligned?(bits_per_level + pe_size)(offset(base′))
------------------------------------------------------------------------------
 {1}  aligned?(pe_size)
                (base′`offset + shift_bits_left(cut_bits(offset(addr′), −1 × bits_per_level × lvl′ − bits_per_level + bus_
```

Expanding the definition of min,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_pe_aligned.1.1.1`.

`xlat_idx_pe_aligned.1.1.2:`

```
{-1}  pe_size ≤ pe_size ⊃
       aligned?(pe_size)
                (shift_bits_left(cut_bits(offset(addr′),
                                          bus_width − (lvl′ + 1) × bits_per_level,
                                          bits_per_level),
                                 pe_size))
{-2}  −1 × (bits_per_level × lvl′) − bits_per_level + bus_width ≥ 0
{-3}  lvl′ < max_level
{-4}  Mem?(base′`type_of)
{-5}  0 ≤ base′`offset
{-6}  base′`offset < max_linear_offset
{-7}  Mem?(addr′`type_of)
{-8}  0 ≤ addr′`offset
{-9}  addr′`offset < max_linear_offset
{-10} aligned?(bits_per_level + pe_size)(offset(base′))
------------------------------------------------------------------------------
 {1}  aligned?(pe_size)
                (shift_bits_left(cut_bits(offset(addr′), 22 − 10 × lvl′, 10), pe_size))
 {2}  aligned?(pe_size)
                (base′`offset + shift_bits_left(cut_bits(offset(addr′), −1 × bits_per_level × lvl′ − bits_per_level + bus_
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_pe_aligned.1.1.2`.

`xlat_idx_pe_aligned.1.1.3:`

```
{-1}   pe_size ≤ pe_size ⊃
         aligned?(pe_size)
                  (shift_bits_left(cut_bits(offset(addr′),
                                            bus_width − (lvl′ + 1) × bits_per_level,
                                            bits_per_level),
                                   pe_size))
{-2}   −1 × (bits_per_level × lvl′) − bits_per_level + bus_width ≥ 0
{-3}   lvl′ < max_level
{-4}   Mem?(base′‘type_of)
{-5}   0 ≤ base′‘offset
{-6}   base′‘offset < max_linear_offset
{-7}   Mem?(addr′‘type_of)
{-8}   0 ≤ addr′‘offset
{-9}   addr′‘offset < max_linear_offset
{-10}  aligned?(bits_per_level + pe_size)(offset(base′))
-------------------------------------------------------------------------------
 {1}   22 − 10 × lvl′ ≥ 0
 {2}   aligned?(pe_size)
                  (base′‘offset + shift_bits_left(cut_bits(offset(addr′),  −1 × bits_per_level × lvl′ − bits_per_
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_pe_aligned.1.1.3`.

`xlat_idx_pe_aligned.1.1.4:`

```
{-1}   pe_size ≤ pe_size ⊃
         aligned?(pe_size)
                  (shift_bits_left(cut_bits(offset(addr′),
                                            bus_width − (lvl′ + 1) × bits_per_level,
                                            bits_per_level),
                                   pe_size))
{-2}   −1 × (bits_per_level × lvl′) − bits_per_level + bus_width ≥ 0
{-3}   lvl′ < max_level
{-4}   Mem?(base′‘type_of)
{-5}   0 ≤ base′‘offset
{-6}   base′‘offset < max_linear_offset
{-7}   Mem?(addr′‘type_of)
{-8}   0 ≤ addr′‘offset
{-9}   addr′‘offset < max_linear_offset
{-10}  aligned?(bits_per_level + pe_size)(offset(base′))
-------------------------------------------------------------------------------
 {1}   aligned?(10 + pe_size)(offset(base′))
 {2}   aligned?(pe_size)
                  (base′‘offset + shift_bits_left(cut_bits(offset(addr′),  −1 × bits_per_level × lvl′ − bits_per_
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_pe_aligned.1.1.4`.

`xlat_idx_pe_aligned.1.2:`

{-1} $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width} \geq 0$
{-2} $\text{lvl}' < \text{max\_level}$
{-3} Mem?(base′`type_of)
{-4} $0 \leq$ base′`offset
{-5} base′`offset $<$ max_linear_offset
{-6} Mem?(addr′`type_of)
{-7} $0 \leq$ addr′`offset
{-8} addr′`offset $<$ max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base′))

{1} $22 - 10 \times \text{lvl}' \geq 0$
{2} aligned?(pe_size)
(offset
(base′ + shift_bits_left(cut_bits(offset(addr′), $-1 \times$ bits_per_level $\times$ lvl′ − bits_per_level + bus_wi

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_pe_aligned.1.2`.
`xlat_idx_pe_aligned.2:`

{-1} $\text{lvl}' < \text{max\_level}$
{-2} Mem?(base′`type_of)
{-3} $0 \leq$ base′`offset
{-4} base′`offset $<$ max_linear_offset
{-5} Mem?(addr′`type_of)
{-6} $0 \leq$ addr′`offset
{-7} addr′`offset $<$ max_linear_offset
{-8} aligned?(bits_per_level + pe_size)(offset(base′))

{1} $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width} \geq 0$
{2} aligned?(pe_size)
(offset
(base′ + shift_bits_left(cut_bits(offset(addr′), $-1 \times$ bits_per_level $\times$ lvl′ − bits_per_level + bus_wi

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_pe_aligned.2`.
Q.E.D.

C.114.6 Linear_Memory.xlat_idx_memory_address

Terse proof for `xlat_idx_memory_address`.
`xlat_idx_memory_address:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
aligned?(bits_per_level + pe_size)(offset(base)) ⊃
xlat_idx(lvl, base, addr) $<$ max_linear

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: Mem + $<$ max_linear max_linear_offset bits_per_level pe_size max_level bus_width
Expanding the definition of xlat_idx,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma aligned_add_below,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using shift_bits_left_less, matching in *,

This completes the proof of `xlat_idx_memory_address`.
Q.E.D.

C.114.7 Linear_Memory.xlat_idx_memory_address2

Terse proof for `xlat_idx_memory_address2`.
`xlat_idx_memory_address2:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
aligned?(bits_per_level + pe_size)(offset(base)) ⊃
0 ≤ offset(xlat_idx(lvl, base, addr))

Expanding the definition of xlat_idx,
Repeatedly Skolemizing and flattening,
Adding type constraints for shift_bits_left(cut_bits(offset(addr!1), -1 * (lvl!1 * bits_per_level) - bits_per_level + bus_width, bits_per_level), pe_size),
we get 2 subgoals:
`xlat_idx_memory_address2.1:`

{-1} shift_bits_left(cut_bits(offset(addr′),
−1 × (lvl′ × bits_per_level) − bits_per_level + bus_width,
bits_per_level),
pe_size)
≥ 0
{-2} lvl′ < max_level
{-3} Mem?(base′‘type_of)
{-4} 0 ≤ base′‘offset
{-5} base′‘offset < max_linear_offset
{-6} Mem?(addr′‘type_of)
{-7} 0 ≤ addr′‘offset
{-8} addr′‘offset < max_linear_offset
{-9} aligned?(bits_per_level + pe_size)(offset(base′))

{1} 0 ≤
offset
(base′ + shift_bits_left(cut_bits(offset(addr′), −1 × lvl′ × bits_per_level − bits_per_level + bus_w

Expanding the definition of +,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_memory_address2.1`.
`xlat_idx_memory_address2.2:`

{-1} lvl′ < max_level
{-2} Mem?(base′‘type_of)
{-3} 0 ≤ base′‘offset
{-4} base′‘offset < max_linear_offset
{-5} Mem?(addr′‘type_of)
{-6} 0 ≤ addr′‘offset
{-7} addr′‘offset < max_linear_offset
{-8} aligned?(bits_per_level + pe_size)(offset(base′))

{1} −1 × (lvl′ × bits_per_level) − bits_per_level + bus_width ≥ 0
{2} 0 ≤
offset
(base′ + shift_bits_left(cut_bits(offset(addr′), −1 × lvl′ × bits_per_level − bits_per_level + bus_w

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Expanding the definition of bus_width,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_memory_address2.2`.

Q.E.D.

C.114.8 Linear_Memory.xlat_ofs_memory_address

Terse proof for `xlat_ofs_memory_address`.

`xlat_ofs_memory_address:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
aligned?(bus_width − (lvl + 1) × bits_per_level)(offset(base)) ⊃
xlat_ofs(lvl, base, addr) < max_linear

Installing automatic rewrites from: Mem + < max_linear max_linear_offset bits_per_level pe_size max_level bus_width

Repeatedly Skolemizing and flattening,

Expanding the definition of xlat_ofs,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma aligned_add_below,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_memory_address`.

Q.E.D.

C.114.9 Linear_Memory.xlat_ofs_memory_address2

Terse proof for `xlat_ofs_memory_address2`.

`xlat_ofs_memory_address2:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G):
aligned?(bus_width − (lvl + 1) × bits_per_level)(offset(base)) ⊃
0 ≤ offset(xlat_ofs(lvl, base, addr))

Expanding the definition of xlat_ofs,

Expanding the definition of +,

Repeatedly Skolemizing and flattening,

Using lemma cut_bits_below,

we get 2 subgoals:

`xlat_ofs_memory_address2.1:`

{-1} cut_bits(offset(addr′), 0,
$-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width}$)
$<$ expt(2, $-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width}$)
{-2} $\text{lvl}' < \text{max\_level}$
{-3} Mem?(base′‘type_of)
{-4} $0 \leq$ base′‘offset
{-5} base′‘offset $<$ max_linear_offset
{-6} Mem?(addr′‘type_of)
{-7} $0 \leq$ addr′‘offset
{-8} addr′‘offset $<$ max_linear_offset
{-9} aligned?($-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width}$)
(offset(base′))

{1} $0 \leq$
base′‘offset + cut_bits(offset(addr′), 0, $-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width}$)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_ofs_memory_address2.1`.
`xlat_ofs_memory_address2.2:`

{-1} $\text{lvl}' < \text{max\_level}$
{-2} Mem?(base′‘type_of)
{-3} $0 \leq$ base′‘offset
{-4} base′‘offset $<$ max_linear_offset
{-5} Mem?(addr′‘type_of)
{-6} $0 \leq$ addr′‘offset
{-7} addr′‘offset $<$ max_linear_offset
{-8} aligned?($-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width}$)
(offset(base′))

{1} $-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width} \geq 0$
{2} $0 \leq$
base′‘offset + cut_bits(offset(addr′), 0, $-1 \times \text{lvl}' \times \text{bits\_per\_level} - \text{bits\_per\_level} + \text{bus\_width}$)

Keeping (-1 1) and hiding *,
Installing automatic rewrites from: bus_width
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_ofs_memory_address2.2`.
Q.E.D.

C.114.10 Linear_Memory.raise_fault_TCC1

Terse proof for `raise_fault_TCC1`.
`raise_fault_TCC1:`

{1} ∀ (access: Memory_access, pfa: Memory_Address_4G):
(Read ∈ (singleton[Memory_access](access) ∪ {Read}))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `raise_fault_TCC1`.
Q.E.D.

C.114.11 Linear_Memory.translate_TCC1

Terse proof for `translate_TCC1`.

`translate_TCC1`:

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
pe_addr: Address):
pe_addr = xlat_idx(lvl, base, addr) ⊃
(∀ (pe: (range_pt(lvl))):
accessible?(pe, access) ∧
privileged?(pe, priv) ∧ present?(pe) ∧ paging_type?(lvl, pe)
⊃ range_pt(lvl)(set_reference(pe, access)))

Repeatedly Skolemizing and flattening,

Rewriting using set_reference_range, matching in *,

This completes the proof of `translate_TCC1`.

Q.E.D.

C.114.12 Linear_Memory.translate_memory_address

Terse proof for `translate_memory_address`.

`translate_memory_address`:

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
s: Linear_memory):
aligned?(bits_per_level + pe_size)(offset(base)) ∧
OK?(translate(lvl, base, addr, access, priv)(s))
⊃ data(translate(lvl, base, addr, access, priv)(s))`2 < max_linear

Expanding the definition of translate,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (##! <! ok_result! exception_result! fatal_result! raise_fault! max_linear! Mem!)

Case splitting on OK?(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1)) AND present?(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))),

we get 2 subgoals:

translate_memory_address.1:

```
{-1}   OK?(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))(s′)) ∧
        present?(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                                                           (s′)))
{-2}   lvl′ < max_level
{-3}   Mem?(base′`type_of)
{-4}   0 ≤ base′`offset
{-5}   base′`offset < max_linear_offset
{-6}   Mem?(addr′`type_of)
{-7}   0 ≤ addr′`offset
{-8}   addr′`offset < max_linear_offset
{-9}   aligned?(bits_per_level + pe_size)(offset(base′))
{-10}  OK?((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
              (λ (pe: (range_pt(lvl′))):
                 IF paging_type?(lvl′, pe)
                   THEN IF present?(pe)
                          THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                                 THEN write_data(pm, paging_data_type(lvl′))
                                                 (xlat_idx(lvl′, base′, addr′),
                                                   set_reference(pe, access′))
                                                 ## ok_result(is_leaf?(pe), base(pe))
                                 ELSE raise_fault(priv′, access′, TRUE, addr′)
                                 ENDIF
                          ELSE raise_fault(priv′, access′, FALSE, addr′)
                          ENDIF
                   ELSE fatal_result
                   ENDIF))
              (s′))
-------------------------------------------------------------------------------
{1}    data((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
              (λ (pe: (range_pt(lvl′))):
                 IF paging_type?(lvl′, pe)
                   THEN IF present?(pe)
                          THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                                 THEN write_data(pm, paging_data_type(lvl′))
                                                 (xlat_idx(lvl′, base′, addr′),
                                                   set_reference(pe, access′))
                                                 ## ok_result(is_leaf?(pe), base(pe))
                                 ELSE raise_fault(priv′, access′, TRUE, addr′)
                                 ENDIF
                          ELSE raise_fault(priv′, access′, FALSE, addr′)
                          ENDIF
                   ELSE fatal_result
                   ENDIF))
              (s′))`2
        < max_linear
```

Adding type constraints for base(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))),

we get 2 subgoals:

translate_memory_address.1.1:

```
{-1}  Mem?(base(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                                (s′)))‘type_of)
{-2}  0 ≤
       base(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                           (s′)))‘offset
{-3}  base(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                           (s′)))‘offset
       < max_linear_offset
{-4}  OK?(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))(s′)) ∧
       present?(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                                (s′)))
{-5}  lvl′ < max_level
{-6}  Mem?(base′‘type_of)
{-7}  0 ≤ base′‘offset
{-8}  base′‘offset < max_linear_offset
{-9}  Mem?(addr′‘type_of)
{-10} 0 ≤ addr′‘offset
{-11} addr′‘offset < max_linear_offset
{-12} aligned?(bits_per_level + pe_size)(offset(base′))
{-13} OK?((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                         THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                                 THEN write_data(pm, paging_data_type(lvl′))
                                                 (xlat_idx(lvl′, base′, addr′),
                                                   set_reference(pe, access′))
                                                 ## ok_result(is_leaf?(pe), base(pe))
                                 ELSE raise_fault(priv′, access′, TRUE, addr′)
                                 ENDIF
                         ELSE raise_fault(priv′, access′, FALSE, addr′)
                         ENDIF
                 ELSE fatal_result
                 ENDIF))
            (s′))
-------------------------------------------------------------------------------
{1}   data((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                         THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                                 THEN write_data(pm, paging_data_type(lvl′))
                                                 (xlat_idx(lvl′, base′, addr′),
                                                   set_reference(pe, access′))
                                                 ## ok_result(is_leaf?(pe), base(pe))
                                 ELSE raise_fault(priv′, access′, TRUE, addr′)
                                 ENDIF
                         ELSE raise_fault(priv′, access′, FALSE, addr′)
                         ENDIF
                 ELSE fatal_result
                 ENDIF))
            (s′))‘2
       < max_linear
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_memory_address.1.1`.
`translate_memory_address.1.2`:

```
{-1}  OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
       present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                                                          (s')))
{-2}  lvl' < max_level
{-3}  Mem?(base'`type_of)
{-4}  0 ≤ base'`offset
{-5}  base'`offset < max_linear_offset
{-6}  Mem?(addr'`type_of)
{-7}  0 ≤ addr'`offset
{-8}  addr'`offset < max_linear_offset
{-9}  aligned?(bits_per_level + pe_size)(offset(base'))
{-10} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
            (λ (pe: (range_pt(lvl'))):
               IF paging_type?(lvl', pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                               THEN write_data(pm, paging_data_type(lvl'))
                                               (xlat_idx(lvl', base', addr'),
                                                 set_reference(pe, access'))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv', access', TRUE, addr')
                             ENDIF
                      ELSE raise_fault(priv', access', FALSE, addr')
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s'))
-------------------------------------------------------------------------------
{1}   present?(data[Physical_memory, ((range_pt(lvl')))]
                 (read_data[Physical_memory, ((range_pt(lvl')))]
                           (pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                           (s')))
{2}   data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
            (λ (pe: (range_pt(lvl'))):
               IF paging_type?(lvl', pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                               THEN write_data(pm, paging_data_type(lvl'))
                                               (xlat_idx(lvl', base', addr'),
                                                 set_reference(pe, access'))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv', access', TRUE, addr')
                             ENDIF
                      ELSE raise_fault(priv', access', FALSE, addr')
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s'))`2
      < max_linear
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `translate_memory_address.1.2`.

`translate_memory_address.2`:

```
{-1}  lvl′ < max_level
{-2}  Mem?(base′`type_of)
{-3}  0 ≤ base′`offset
{-4}  base′`offset < max_linear_offset
{-5}  Mem?(addr′`type_of)
{-6}  0 ≤ addr′`offset
{-7}  addr′`offset < max_linear_offset
{-8}  aligned?(bits_per_level + pe_size)(offset(base′))
{-9}  OK?((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s′))
-------------------------------------------------------------------------------
{1}  OK?(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))(s′)) ∧
      present?(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                    (s′)))
{2}  data((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s′))`2
      < max_linear
```

Hiding formulas: 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_memory_address.2`.

Q.E.D.

C.114.13 Linear_Memory.translate_memory_address2

Terse proof for `translate_memory_address2`.

`translate_memory_address2:`

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
s: Linear_memory):
aligned?(bits_per_level + pe_size)(offset(base)) ∧
OK?(translate(lvl, base, addr, access, priv)(s))
⊃ 0 ≤ offset(data(translate(lvl, base, addr, access, priv)(s))‘2)

Expanding the definition of translate,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (##! <! ok_result! exception_result! fatal_result! raise_fault! max_linear! Mem!)

Case splitting on OK?(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1)) AND present?(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))),

we get 2 subgoals:

`translate_memory_address2.1:`

```
{-1}  OK?(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))(s′)) ∧
       present?(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                                                          (s′)))
{-2}  lvl′ < max_level
{-3}  Mem?(base′`type_of)
{-4}  0 ≤ base′`offset
{-5}  base′`offset < max_linear_offset
{-6}  Mem?(addr′`type_of)
{-7}  0 ≤ addr′`offset
{-8}  addr′`offset < max_linear_offset
{-9}  aligned?(bits_per_level + pe_size)(offset(base′))
{-10} OK?((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm, paging_data_type(lvl′))
                                                   (xlat_idx(lvl′, base′, addr′),
                                                     set_reference(pe, access′))
                                                   ## ok_result(is_leaf?(pe), base(pe))
                               ELSE raise_fault(priv′, access′, TRUE, addr′)
                               ENDIF
                        ELSE raise_fault(priv′, access′, FALSE, addr′)
                        ENDIF
                 ELSE fatal_result
                 ENDIF))
           (s′))
-------------------------------------------------------------------------------
{1}   0 ≤
       offset
         (data((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
                  (λ (pe: (range_pt(lvl′))):
                     IF paging_type?(lvl′, pe)
                       THEN IF present?(pe)
                              THEN IF accessible?(pe, access′) ∧ privi-
      leged?(pe, priv′)
                                     THEN write_data(pm, paging_data_type(lvl′))
                                                         (xlat_idx(lvl′, base′, addr′),
                                                           set_reference(pe, access′))
                                                         ## ok_result(is_leaf?(pe), base(pe))
                                     ELSE raise_fault(priv′, access′, TRUE, addr′)
                                     ENDIF
                              ELSE raise_fault(priv′, access′, FALSE, addr′)
                              ENDIF
                       ELSE fatal_result
                       ENDIF))
                 (s′))`2)
```

Adding type constraints for base(data(read_data(pm, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, addr!1))(s!1))),

we get 2 subgoals:

translate_memory_address2.1.1:

```
{-1}  Mem?(base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                                                    (s')))`type_of)
{-2}  0 ≤
       base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                                                  (s')))`offset
{-3}  base(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                                                  (s')))`offset
       < max_linear_offset
{-4}  OK?(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))(s')) ∧
       present?(data(read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr'))
                                                  (s')))
{-5}  lvl' < max_level
{-6}  Mem?(base'`type_of)
{-7}  0 ≤ base'`offset
{-8}  base'`offset < max_linear_offset
{-9}  Mem?(addr'`type_of)
{-10} 0 ≤ addr'`offset
{-11} addr'`offset < max_linear_offset
{-12} aligned?(bits_per_level + pe_size)(offset(base'))
{-13} OK?((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
            (λ (pe: (range_pt(lvl'))):
               IF paging_type?(lvl', pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access') ∧ privileged?(pe, priv')
                               THEN write_data(pm, paging_data_type(lvl'))
                                                  (xlat_idx(lvl', base', addr'),
                                                    set_reference(pe, access'))
                                 ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv', access', TRUE, addr')
                             ENDIF
                      ELSE raise_fault(priv', access', FALSE, addr')
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s'))
-------------------------------------------------------------------------------
{1}   0 ≤
       offset
         (data((read_data(pm, paging_data_type(lvl'))(xlat_idx(lvl', base', addr')) ##
                  (λ (pe: (range_pt(lvl'))):
                     IF paging_type?(lvl', pe)
                       THEN IF present?(pe)
                              THEN IF accessible?(pe, access') ∧ privi-
leged?(pe, priv')
                                     THEN write_data(pm, paging_data_type(lvl'))
                                                  (xlat_idx(lvl', base', addr'),
                                                    set_reference(pe, access'))
                                       ## ok_result(is_leaf?(pe), base(pe))
                                   ELSE raise_fault(priv', access', TRUE, addr')
                                   ENDIF
                            ELSE raise_fault(priv', access', FALSE, addr')
                            ENDIF
                       ELSE fatal_result
                       ENDIF))
                (s'))`2)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_memory_address2.1.1`.

`translate_memory_address2.1.2:`

```
{-1}  OK?(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))(s′)) ∧
       present?(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                                                          (s′)))
{-2}  lvl′ < max_level
{-3}  Mem?(base′‘type_of)
{-4}  0 ≤ base′‘offset
{-5}  base′‘offset < max_linear_offset
{-6}  Mem?(addr′‘type_of)
{-7}  0 ≤ addr′‘offset
{-8}  addr′‘offset < max_linear_offset
{-9}  aligned?(bits_per_level + pe_size)(offset(base′))
{-10} OK?((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s′))
-------------------------------------------------------------------------------
{1}   present?(data[Physical_memory, ((range_pt(lvl′)))]
                 (read_data[Physical_memory, ((range_pt(lvl′)))]
                           (pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                           (s′)))
{2}   0 ≤
       offset
         (data((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
                 (λ (pe: (range_pt(lvl′))):
                    IF paging_type?(lvl′, pe)
                      THEN IF present?(pe)
                             THEN IF accessible?(pe, access′) ∧ privi-
leged?(pe, priv′)
                                    THEN write_data(pm, paging_data_type(lvl′))
                                                    (xlat_idx(lvl′, base′, addr′),
                                                      set_reference(pe, access′))
                                         ## ok_result(is_leaf?(pe), base(pe))
                                  ELSE raise_fault(priv′, access′, TRUE, addr′)
                                  ENDIF
                           ELSE raise_fault(priv′, access′, FALSE, addr′)
                           ENDIF
                      ELSE fatal_result
                      ENDIF))
               (s′))‘2)
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `translate_memory_address2.1.2`.
`translate_memory_address2.2`:

```
{-1}  lvl′ < max_level
{-2}  Mem?(base′`type_of)
{-3}  0 ≤ base′`offset
{-4}  base′`offset < max_linear_offset
{-5}  Mem?(addr′`type_of)
{-6}  0 ≤ addr′`offset
{-7}  addr′`offset < max_linear_offset
{-8}  aligned?(bits_per_level + pe_size)(offset(base′))
{-9}  OK?((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
               ELSE fatal_result
               ENDIF))
           (s′))
-------------------------------------------------------------------------------
{1}   OK?(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))(s′)) ∧
       present?(data(read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′))
                     (s′)))
{2}   0 ≤
       offset
         (data((read_data(pm, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
                  (λ (pe: (range_pt(lvl′))):
                     IF paging_type?(lvl′, pe)
                       THEN IF present?(pe)
                              THEN IF accessible?(pe, access′) ∧ privi-
      leged?(pe, priv′)
                                     THEN write_data(pm, paging_data_type(lvl′))
                                                     (xlat_idx(lvl′, base′, addr′),
                                                       set_reference(pe, access′))
                                          ## ok_result(is_leaf?(pe), base(pe))
                                   ELSE raise_fault(priv′, access′, TRUE, addr′)
                                   ENDIF
                            ELSE raise_fault(priv′, access′, FALSE, addr′)
                            ENDIF
                     ELSE fatal_result
                     ENDIF))
                (s′))`2)
```

Hiding formulas: 2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_memory_address2.2`.
Q.E.D.

C.114.14 Linear_Memory.translate_result_leaf_TCC1

Terse proof for `translate_result_leaf_TCC1`.

`translate_result_leaf_TCC1:`

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
        s: Linear_memory):
       aligned?(bits_per_level + pe_size)(offset(base)) ∧
        OK?(translate(lvl, base, addr, access, priv)(s)) ∧
         data(translate(lvl, base, addr, access, priv)(s))`1
        ⊃
        offset
          (data[Physical_memory, [bool, Address]]
                (translate(lvl, base, addr, access, priv)(s))`2)
         ≥ 0
```

Repeatedly Skolemizing and flattening,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_leaf_TCC1`.

Q.E.D.

C.114.15 Linear_Memory.translate_result_leaf_TCC2

Terse proof for `translate_result_leaf_TCC2`.

`translate_result_leaf_TCC2:`

```
{1}  ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
        s: Linear_memory):
       aligned?(bits_per_level + pe_size)(offset(base)) ∧
        OK?(translate(lvl, base, addr, access, priv)(s)) ∧
         data(translate(lvl, base, addr, access, priv)(s))`1
        ⊃ bus_width − bits_per_level × (lvl + 1) ≥ 0
```

Repeatedly Skolemizing and flattening,

Keeping (-1 1) and hiding *,

Installing automatic rewrites from: bus_width

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_result_leaf_TCC2`.

Q.E.D.

C.114.16 Linear_Memory.translate_result_leaf

Terse proof for `translate_result_leaf`.

`translate_result_leaf`:

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
s: Linear_memory):
aligned?(bits_per_level + pe_size)(offset(base)) ∧
OK?(translate(lvl, base, addr, access, priv)(s)) ∧
data(translate(lvl, base, addr, access, priv)(s))`1
⊃
aligned?(bus_width − bits_per_level × (lvl + 1))
(offset(data(translate(lvl, base, addr, access, priv)(s))`2))

Repeatedly Skolemizing and flattening,

Expanding the definition of translate,

Installing automatic rewrites from: (##! ok_result! exception_result! fatal_result! raise_fault! is_leaf?! bus_width! pe_size! bits_per_level! base! paging_tape?! pdir_entry?! ptab_entry?! max_level!)

Expanding the definition of paging_type?,

Expanding the definition of pdir_entry?,

Expanding the definition of ptab_entry?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_leaf`.

Q.E.D.

C.114.17 Linear_Memory.translate_result_no_leaf_TCC1

Terse proof for `translate_result_no_leaf_TCC1`.

`translate_result_no_leaf_TCC1`:

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
s: Linear_memory):
aligned?(bits_per_level + pe_size)(offset(base)) ∧
OK?(translate(lvl, base, addr, access, priv)(s)) ∧
¬ data(translate(lvl, base, addr, access, priv)(s))`1
⊃
offset
(data[Physical_memory, [bool, Address]]
(translate(lvl, base, addr, access, priv)(s))`2)
≥ 0

Repeatedly Skolemizing and flattening,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_no_leaf_TCC1`.

Q.E.D.

C.114.18 Linear_Memory.translate_result_no_leaf

Terse proof for `translate_result_no_leaf`.

`translate_result_no_leaf`:

{1} ∀ (lvl: Level, base, addr: Memory_Address_4G, access: Memory_access, priv: Memory_privilege,
s: Linear_memory):
aligned?(bits_per_level + pe_size)(offset(base)) ∧
OK?(translate(lvl, base, addr, access, priv)(s)) ∧
¬ data(translate(lvl, base, addr, access, priv)(s))‘1
⊃
aligned?(bits_per_level + pe_size)
(offset(data(translate(lvl, base, addr, access, priv)(s))‘2))

Repeatedly Skolemizing and flattening,

Expanding the definition of translate,

Installing automatic rewrites from: (##! ok_result! exception_result! fatal_result! raise_fault! is_leaf?! bus_width! pe_size! bits_per_level! base! paging_tape?! pdir_entry?! ptab_entry?! max_level!)

Expanding the definition of paging_type?,

Expanding the definition of pdir_entry?,

Expanding the definition of ptab_entry?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_no_leaf`.

Q.E.D.

C.114.19 Linear_Memory.linear_resolve_TCC1

Terse proof for `linear_resolve_TCC1`.

`linear_resolve_TCC1`:

{1} ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
priv = segment_to_priv(cs) ⊃
(∀ (pdbr: Pdbr_type, lvl: nat): lvl = pdir_lvl ⊃ lvl < max_level)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_TCC1`.

Q.E.D.

C.114.20 Linear_Memory.linear_resolve_TCC2

Terse proof for `linear_resolve_TCC2`.

`linear_resolve_TCC2:`

```
{1}  ∀ (addr: Memory_Address_4G, access: Memory_access, cs: Segment_Reg_type,
          priv: Memory_privilege):
       priv = segment_to_priv(cs) ⊃
        (∀ (pdbr: Pdbr_type, lvl: nat):
            lvl = pdir_lvl ⊃
             (∀ (x1: Physical_memory):
                 every[Physical_memory, [bool, Address]]
                      (λ (x: Physical_memory): TRUE,
                        λ (t: [bool, Address]):
                          Mem?(t`2`type_of) ∧
                           (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                            IF t`1
                               THEN aligned?(−1 × lvl × bits_per_level − bits_per_level + bus_width)
                                             (offset(t`2))
                            ELSE aligned?(bits_per_level + pe_size)(offset(t`2))
                            ENDIF)
                      (translate(lvl, pdbr`base_addr, addr, access, priv)(x1))))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Using lemma translate_result_leaf,
Using lemma translate_result_no_leaf,
Using lemma translate_memory_address,
Using lemma translate_memory_address2,
Installing automatic rewrites from: max_linear Mem < bits_per_level pe_size
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_TCC2`.
Q.E.D.

C.114.21 Linear_Memory.linear_resolve_TCC3

Terse proof for `linear_resolve_TCC3`.

`linear_resolve_TCC3:`

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
       priv = segment_to_priv(cs) ⊃
        (∀ (pdbr: Pdbr_type, lvl: nat):
            lvl = pdir_lvl ⊃ (∀ (leaf: bool, b: Memory_Address_4G): leaf ⊃ off-
     set(b) ≥ 0))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: bus_width bits_per_level max_level
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_TCC3`.
Q.E.D.

C.114.22 Linear_Memory.linear_resolve_TCC4

Terse proof for `linear_resolve_TCC4`.

`linear_resolve_TCC4`:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
       priv = segment_to_priv(cs) ⊃
        (∀ (pdbr: Pdbr_type, lvl: nat):
             lvl = pdir_lvl ⊃
              (∀ (leaf: bool, b: Memory_Address_4G):
                   leaf ⊃ bus_width − bits_per_level × (lvl + 1) ≥ 0))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: bus_width bits_per_level max_level pdir_lvl
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `linear_resolve_TCC4`.
Q.E.D.

C.114.23 Linear_Memory.linear_resolve_TCC5

Terse proof for `linear_resolve_TCC5`.

`linear_resolve_TCC5`:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
       priv = segment_to_priv(cs) ⊃
        (∀ (pdbr: Pdbr_type, lvl: nat):
             lvl = pdir_lvl ⊃
              (∀ (leaf: bool, b: Memory_Address_4G): ¬ leaf ⊃ offset(b) ≥ 0))
```

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_TCC5`.
Q.E.D.

C.114.24 Linear_Memory.linear_resolve_TCC6

Terse proof for `linear_resolve_TCC6`.

`linear_resolve_TCC6`:

```
{1}  ∀ (addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
       priv = segment_to_priv(cs) ⊃
        (∀ (pdbr: Pdbr_type, lvl: nat):
             lvl = pdir_lvl ⊃
              (∀ (leaf: bool,
                  base:
                    {b: Memory_Address_4G |
                                IF leaf
                                  THEN aligned?(bus_width − bits_per_level × (lvl + 1))
                                                (offset(b))
                                ELSE aligned?(bits_per_level + pe_size)(offset(b))
                                ENDIF}):
                 ¬ leaf ⊃ (∀ (lvl1: nat): lvl1 = ptab_lvl ⊃ lvl1 < max_level)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_resolve_TCC6`.

Q.E.D.

C.114.25 Linear_Memory.linear_resolve_TCC7

Terse proof for `linear_resolve_TCC7`.

`linear_resolve_TCC7`:

{1} ∀ (addr: Memory_Address_4G, access: Memory_access, cs: Segment_Reg_type,
priv: Memory_privilege):
priv = segment_to_priv(cs) ⊃
(∀ (pdbr: Pdbr_type, lvl: nat):
lvl = pdir_lvl ⊃
(∀ (leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(bus_width − bits_per_level × (lvl + 1))
(offset(b))
ELSE aligned?(bits_per_level + pe_size)(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (lvl1: nat):
lvl1 = ptab_lvl ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate(lvl1, base, addr, access, priv)(x_1))))))

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Installing automatic rewrites from: max_linear Mem < bits_per_level pe_size

Using lemma translate_result_leaf,

Using lemma translate_result_no_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_TCC7`.

Q.E.D.

C.114.26 Linear_Memory.linear_resolve_TCC8

Terse proof for `linear_resolve_TCC8`.

`linear_resolve_TCC8`:

{1} ∀ (leaf: bool, addr: Memory_Address_4G, cs: Segment_Reg_type, priv: Memory_privilege):
priv = segment_to_priv(cs) ⊃
(∀ (pdbr: Pdbr_type, lvl: nat):
lvl = pdir_lvl ⊃
(∀ (*b*: Memory_Address_4G):
leaf ⊃
−1 × lvl × bits_per_level − bits_per_level + bus_width ≥ 0))

Repeatedly Skolemizing and flattening,

Keeping (-9 1) and hiding *,

Installing automatic rewrites from: bus_width

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_TCC8`.

Q.E.D.

C.114.27 Linear_Memory.linear_resolve_memory_address

Terse proof for `linear_resolve_memory_address`.

`linear_resolve_memory_address`:

{1} ∀ (access: Memory_access, addr: Memory_Address_4G, x_1: Linear_memory):
every[Linear_memory, Address]
(λ (*x*: Linear_memory): TRUE,
λ (*x*: Address):
Mem?(*x*`type_of) ∧ 0 ≤ *x*`offset ∧ *x*`offset < max_linear_offset)
(linear_resolve(addr, access)(x_1))

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Expanding the definition of ok_result,

Installing automatic rewrites from: max_linear Mem < bits_per_level bus_width min_linear Address_Helpers.<= max_level pe_size xlat_ofs_memory_address! pdir_lvl ptab_lvl translate_result_leaf translate_result_no_leaf translate_memory_address! translate_memory_address2! xlat_ofs_memory_address2! translate_result_leaf translate_result_no_leaf

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 6 subgoals:

`linear_resolve_memory_address.1`:

{-1} Mem?(addr′`type_of)
{-2} $0 \leq$ addr′`offset
{-3} addr′`offset $<$ max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x_1'))
{-5} OK?(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
{-6} OK?(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1')))`base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))
{-7} data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1')))`base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))`1

{1} xlat_ofs(0,
data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))`base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))`2,
addr′)`offset
$<$ max_linear_offset

Using lemma xlat_ofs_memory_address,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.1`.

`linear_resolve_memory_address.2:`

{-1} Mem?(addr′‘type_of)
{-2} $0 \leq$ addr′‘offset
{-3} addr′‘offset $<$ max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x'_1))
{-5} OK?(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)(x'_1))))
{-6} OK?(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x'_1))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x'_1))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x'_1))))))
{-7} data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x'_1))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x'_1))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x'_1))))))‘1

{1} $0 \leq$
xlat_ofs(0,
data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x'_1))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x'_1))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x'_1))))))‘2,
addr′)‘offset

Using lemma xlat_ofs_memory_address2,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.2`.

`linear_resolve_memory_address.3`:

```
{-1}  Mem?(addr′`type_of)
{-2}  0 ≤ addr′`offset
{-3}  addr′`offset < max_linear_offset
{-4}  OK?(read_data(pm, segment_reg_data_type)(CS)(x′1))
{-5}  OK?(read_data(pm, pdbr_data_type)(PDBR)
                    (state(read_data(pm, segment_reg_data_type)(CS)(x′1))))
{-6}  OK?(translate(0,
                    data(read_data(pm, pdbr_data_type)(PDBR)
                                   (state(read_data(pm, segment_reg_data_type)(CS)
                                                   (x′1))))`base_addr,
                    addr′, access′,
                    segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x′1))))
                    (state(read_data(pm, pdbr_data_type)(PDBR)
                                   (state(read_data(pm, segment_reg_data_type)(CS)
                                                   (x′1))))))
{-7}  data(translate(0,
                     data(read_data(pm, pdbr_data_type)(PDBR)
                                    (state(read_data(pm, segment_reg_data_type)(CS)
                                                    (x′1))))`base_addr,
                     addr′, access′,
                     segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x′1))))
                     (state(read_data(pm, pdbr_data_type)(PDBR)
                                    (state(read_data(pm, segment_reg_data_type)(CS)
                                                    (x′1))))))`1
-------------------------------------------------------------------------------------------
{1}   Mem?(xlat_ofs(0,
                    data(translate(0,
                                   data(read_data(pm, pdbr_data_type)(PDBR)
                                                  (state(read_data(pm, segment_reg_data_type)
                                                                  (CS)
                                                                  (x′1))))`base_addr,
                                   addr′, access′,
                                   segment_to_priv(data(read_data(pm, segment_reg_data_type)
                                                               (CS)
                                                               (x′1))))
                                   (state(read_data(pm, pdbr_data_type)(PDBR)
                                                  (state(read_data(pm, segment_reg_data_type)
                                                                  (CS)
                                                                  (x′1))))))`2,
                    addr′)`type_of)
```

Using lemma xlat_ofs_memory_address,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.3`.

linear_resolve_memory_address.4:

{-1} Mem?(addr′‘type_of)
{-2} $0 \leq$ addr′‘offset
{-3} addr′‘offset $<$ max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x_1'))
{-5} OK?(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
{-6} OK?(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))
{-7} OK?(translate(1,
data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))‘2,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))))

{1} data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))‘1
{2} xlat_ofs(1,

Using lemma translate_result_no_leaf,

Using lemma translate_result_no_leaf,

Using lemma translate_result_leaf,

Using lemma xlat_ofs_memory_address,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.4`.

```
linear_resolve_memory_address.5:
```

{-1} Mem?(addr′‘type_of)
{-2} 0 ≤ addr′‘offset
{-3} addr′‘offset < max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x_1'))
{-5} OK?(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
{-6} OK?(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))
{-7} OK?(translate(1,
data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))‘2,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))))

{1} data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))‘1
{2} 0 ≤

Using lemma xlat_ofs_memory_address2,

Using lemma translate_result_no_leaf,

Using lemma translate_result_no_leaf,

Using lemma translate_result_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_memory_address.5`.

linear_resolve_memory_address.6:

{-1} Mem?(addr′‘type_of)
{-2} 0 ≤ addr′‘offset
{-3} addr′‘offset < max_linear_offset
{-4} OK?(read_data(pm, segment_reg_data_type)(CS)(x_1'))
{-5} OK?(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
{-6} OK?(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))
{-7} OK?(translate(1,
data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))‘2,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)
(CS)
(x_1'))))))))

{1} data(translate(0,
data(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))‘base_addr,
addr′, access′,
segment_to_priv(data(read_data(pm, segment_reg_data_type)(CS)(x_1'))))
(state(read_data(pm, pdbr_data_type)(PDBR)
(state(read_data(pm, segment_reg_data_type)(CS)
(x_1'))))))‘1
{2} Mem?(xlat_ofs(1,

Using lemma translate_result_no_leaf,
Using lemma translate_result_leaf,
Using lemma translate_result_no_leaf,
Using lemma translate_memory_address,
Using lemma xlat_ofs_memory_address,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_memory_address.6`.
Q.E.D.

C.114.28 Linear_Memory.linear_read_TCC1

Terse proof for `linear_read_TCC1`.
`linear_read_TCC1`:

{1} Mem?(min_linear‘type_of)

Expanding the definition of min_linear,
Rewriting using Mem, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `linear_read_TCC1`.
Q.E.D.

C.114.29 Linear_Memory.linear_read_TCC2

Terse proof for `linear_read_TCC2`.
`linear_read_TCC2`:

{1} Mem?(max_linear‘type_of)

Expanding the definition of max_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of `linear_read_TCC2`.
Q.E.D.

C.114.30 Linear_Memory.linear_read_TCC3

Terse proof for `linear_read_TCC3`.
`linear_read_TCC3`:

{1} $\forall$ (a: Address):
Mem?(type_of(a)) $\wedge$ in_memory(min_linear, max_linear)(a) $\supset$
$0 \leq a$‘offset $\wedge$ a‘offset $<$ max_linear_offset

Expanding the definition of min_linear,
Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of max_linear,
Expanding the definition of max_linear_offset,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_read_TCC3`.
Q.E.D.

C.114.31 Linear_Memory.linear_read_side_effect_TCC1

Terse proof for `linear_read_side_effect_TCC1`.

`linear_read_side_effect_TCC1`:

```
{1}  ∀ (a: Address, bl: list[Byte], s: Linear_memory):
       Mem?(type_of(a)) ∧
        (null?(bl) ∨
           (reg_base(min_linear)(type_of(a)) ≤ a ∧
              a + length(bl) ≤ reg_size(max_linear)(type_of(a))))
        ⊃
        every[[Address, list[Byte]]]
             (λ (t: [Address, list[Byte]]):
                Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
             (split(min_page, a, bl))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Case splitting on bl!1 = null,
we get 2 subgoals:

`linear_read_side_effect_TCC1.1`:

```
{-1}  bl′ = null
{-2}  Mem?(type_of(a′))
{-3}  null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
------------------------------------------------------------------
{1}   every[[Address, list[Byte]]]
           (λ (t: [Address, list[Byte]]):
              Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
           (split(min_page, a′, bl′))
```

Using lemma split_null,
Expanding the definition of every,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_read_side_effect_TCC1.1`.

`linear_read_side_effect_TCC1.2`:

```
{-1}  Mem?(type_of(a′))
{-2}  null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
------------------------------------------------------------------
{1}   bl′ = null
{2}   every[[Address, list[Byte]]]
           (λ (t: [Address, list[Byte]]):
              Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
           (split(min_page, a′, bl′))
```

Using lemma split_range,
Using lemma split_no_null,
Using lemma every_implied,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_read_side_effect_TCC1.2.1:`

```
{-1}  cons?(bl')
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-4}  Mem?(type_of(a'))
{-5}  reg_base(min_linear)(type_of(a')) ≤ a'
{-6}  a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))
|-------
{1}   every((λ (a2: Address, bl2: list[Byte]):
                 cons?(bl2) ∧ a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl')))
           (split(min_page, a', bl'))
{2}   every[[Address, list[Byte]]]
           (λ (t: [Address, list[Byte]]):
                Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
           (split(min_page, a', bl'))
```

Hiding formulas: 2,

Using lemma every_conjunct_left,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_read_side_effect_TCC1.2.1`.

`linear_read_side_effect_TCC1.2.2:`

```
{-1}  cons?(bl')
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-4}  Mem?(type_of(a'))
{-5}  reg_base(min_linear)(type_of(a')) ≤ a'
{-6}  a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))
|-------
{1}   ∀ (t_1: [Address, list[Byte]]):
        cons?(t_1`2) ∧ a' ≤ t_1`1 ∧ t_1`1 + length(t_1`2) ≤ a' + length(bl')
          ⊃ Mem?(t_1`1`type_of) ∧ 0 ≤ t_1`1`offset ∧ t_1`1`offset < max_linear_offset
{2}   every[[Address, list[Byte]]]
           (λ (t: [Address, list[Byte]]):
                Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
           (split(min_page, a', bl'))
```

Hiding formulas: (-2 -3 2),

Trying repeated skolemization, instantiation, and if-lifting,

Expanding the definition of max_linear,

Expanding the definition of Mem,

Expanding the definition of max_linear_offset,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_read_side_effect_TCC1.2.2`.

Q.E.D.

C.115 Proofs for Linear_Memory_Blessing (challenge-linear.pvs)

C.115.1 Linear_Memory_Blessing.PTab_Address_TCC1

Terse proof for `PTab_Address_TCC1`.

`PTab_Address_TCC1`:

{1} ∀ (a: Memory_Address_4G): offset(a) ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `PTab_Address_TCC1`.
Q.E.D.

C.115.2 Linear_Memory_Blessing.pe_in_pdir_range?_TCC1

Terse proof for `pe_in_pdir_range?_TCC1`.

`pe_in_pdir_range?_TCC1`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], linear_range: PRED[Memory_Address_4G],
phy_a: Memory_Address_4G, res: ExprResult[Physical_memory, Pdbr_type]):
OK?(res) ∧ res = read_data(pm_phy, pdbr_data_type)(PDBR)(s) ⊃
(∀ (lin_a: (linear_range)): pdir_lvl[Physical_memory, pm_phy] < max_level)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pe_in_pdir_range?_TCC1`.
Q.E.D.

C.115.3 Linear_Memory_Blessing.pe_in_ptab_range?_TCC1

Terse proof for `pe_in_ptab_range?_TCC1`.

`pe_in_ptab_range?_TCC1`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], linear_range: PRED[Memory_Address_4G],
phy_a: Memory_Address_4G, pde: (pe_in_pdir_range?(s, linear_range)),
lin_a: (linear_range)):
pdir_lvl[Physical_memory, pm_phy] < max_level

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pe_in_ptab_range?_TCC1`.
Q.E.D.

C.115.4 Linear_Memory_Blessing.pe_in_ptab_range?_TCC2

Terse proof for `pe_in_ptab_range?_TCC2`.

pe_in_ptab_range?_TCC2:

```
{1}  ∀ (s: Linear_memory[Physical_memory, pm_phy], linear_range: PRED[Memory_Address_4G],
          phy_a: Memory_Address_4G, pde: (pe_in_pdir_range?(s, linear_range)),
          lin_a: (linear_range),
          res: ExprResult[Physical_memory, ((range_pt(pdir_lvl[Physical_memory, pm_phy])))]):
       OK?(res) ∧ res = read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s) ⊃
        pdir?(data[Physical_memory, ((range_pt(pdir_lvl)))](res))
```

Repeatedly Skolemizing and flattening,
Adding type constraints for data[Physical_memory, ((range_pt(pdir_lvl)))](res!1),
Expanding the definition of range_pt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pdir_lvl,
which is trivially true.
This completes the proof of pe_in_ptab_range?_TCC2.
Q.E.D.

C.115.5 Linear_Memory_Blessing.pe_in_ptab_range?_TCC3

Terse proof for pe_in_ptab_range?_TCC3.
pe_in_ptab_range?_TCC3:

```
{1}  ∀ (s: Linear_memory[Physical_memory, pm_phy], linear_range: PRED[Memory_Address_4G],
          phy_a: Memory_Address_4G, pde: (pe_in_pdir_range?(s, linear_range)),
          lin_a: (linear_range),
          res: ExprResult[Physical_memory, ((range_pt(pdir_lvl[Physical_memory, pm_phy])))]):
       pde_pt?(pdir(data(res))) ∧
        OK?(res) ∧ res = read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s)
        ⊃ ptab_lvl[Physical_memory, pm_phy] < max_level
```

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pe_in_ptab_range?_TCC3.
Q.E.D.

C.115.6 Linear_Memory_Blessing.pe_in_ptab_range?_TCC4

Terse proof for pe_in_ptab_range?_TCC4.
pe_in_ptab_range?_TCC4:

```
{1}  ∀ (s: Linear_memory[Physical_memory, pm_phy], linear_range: PRED[Memory_Address_4G],
          phy_a: Memory_Address_4G, pde: (pe_in_pdir_range?(s, linear_range)),
          lin_a: (linear_range),
          res: ExprResult[Physical_memory, ((range_pt(pdir_lvl[Physical_memory, pm_phy])))]):
       pde_pt?(pdir(data(res))) ∧
        OK?(res) ∧ res = read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s)
        ⊃ present?(data[Physical_memory, ((range_pt(pdir_lvl)))](res))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of present?,

Adding type constraints for data(res!1),
Expanding the definition of range_pt,
Expanding the definition of pdir_lvl,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pe_in_ptab_range?_TCC4`.
Q.E.D.

C.115.7 Linear_Memory_Blessing.pe_in_pt_range?_TCC1

Terse proof for `pe_in_pt_range?_TCC1`.
`pe_in_pt_range?_TCC1`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], lvl: Level, phy_a: Memory_Address_4G):
¬ (lvl = pdir_lvl[Physical_memory, pm_phy] ∧ lvl = ptab_lvl[Physical_memory, pm_phy])

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pe_in_pt_range?_TCC1`.
Q.E.D.

C.115.8 Linear_Memory_Blessing.pe_in_pt_range?_TCC2

Terse proof for `pe_in_pt_range?_TCC2`.
`pe_in_pt_range?_TCC2`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], lvl: Level, phy_a: Memory_Address_4G):
lvl = pdir_lvl[Physical_memory, pm_phy] ∨ lvl = ptab_lvl[Physical_memory, pm_phy]

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pe_in_pt_range?_TCC2`.
Q.E.D.

C.115.9 Linear_Memory_Blessing.pe_in_pt_range_aligned

Terse proof for `pe_in_pt_range_aligned`.
`pe_in_pt_range_aligned`:

{1} ∀ (linear_range: PRED[Memory_Address_4G], lvl: Level,
s: Linear_memory[Physical_memory, pm_phy], phy_a: Memory_Address_4G):
pe_in_pt_range?(s, linear_range, lvl)(phy_a) ⊃ aligned?(pe_size)(offset(phy_a))

Expanding the definition of pe_in_pt_range?,
Expanding the definition of pe_in_ptab_range?,
Expanding the definition of pe_in_pdir_range?,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: pe_size bits_per_level xlat_idx_pe_aligned
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`pe_in_pt_range_aligned.1`:

{-1} $\mathrm{lvl}' < \mathrm{max\_level}$
{-2} Mem?(phy_a′‘type_of)
{-3} $0 \leq$ phy_a′‘offset
{-4} phy_a′‘offset $<$ max_linear_offset
{-5} $\mathrm{lvl}' = \mathrm{pdir\_lvl}$
{-6} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'))
{-7} $\exists$ (lin_a: (linear_range′)):
xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'))‘base_addr, lin_a)
$=$ phy_a′

{1} aligned?(2)(offset(phy_a′))

Repeatedly Skolemizing and flattening,
Adding type constraints for (data(read_data(pm_phy, pdbr_data_type)(PDBR)(s!1))‘base_addr),
Using lemma xlat_idx_pe_aligned,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pe_in_pt_range_aligned.1`.
`pe_in_pt_range_aligned.2`:

{-1} $\mathrm{lvl}' < \mathrm{max\_level}$
{-2} Mem?(phy_a′‘type_of)
{-3} $0 \leq$ phy_a′‘offset
{-4} phy_a′‘offset $<$ max_linear_offset
{-5} $\exists$ (pde: (pe_in_pdir_range?(s', linear_range′))):
$\exists$ (lin_a: (linear_range′)):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s')) $\wedge$
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s')))) $\wedge$
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'))),
lin_a)
$=$ phy_a′

{1} $\mathrm{lvl}' = \mathrm{pdir\_lvl}$
{2} aligned?(2)(offset(phy_a′))

Repeatedly Skolemizing and flattening,
Using lemma xlat_idx_pe_aligned,
Expanding the definition of base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pe_in_pt_range_aligned.2`.
Q.E.D.

C.115.10 Linear_Memory_Blessing.pe_in_pt_address_in_pt

Terse proof for `pe_in_pt_address_in_pt`.
`pe_in_pt_address_in_pt`:

{1} $\forall$ (linear_range: PRED[Memory_Address_4G], lvl: Level,
s: Linear_memory[Physical_memory, pm_phy], base: Memory_Address_4G):
pe_in_pt_range?(s, linear_range, lvl)(base) $\supset$
(address_block(base, expt(2, pe_size)) $\subseteq$ extend [Address, Memory_Address_4G, bool, FALSE](address_i

Repeatedly Skolemizing and flattening,
Expanding the definition of subset?,

Expanding the definition of member,
Repeatedly Skolemizing and flattening,
Expanding the definition of address_in_pt_range?,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`pe_in_pt_address_in_pt.1:`

{-1} $\mathrm{lvl}' < \mathrm{max\_level}$
{-2} $\mathrm{Mem?}(\mathrm{base}'\text{`}\mathrm{type\_of})$
{-3} $0 \leq \mathrm{base}'\text{`}\mathrm{offset}$
{-4} $\mathrm{base}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-5} $\mathrm{pe\_in\_pt\_range?}(s', \mathrm{linear\_range}', \mathrm{lvl}')(\mathrm{base}')$
{-6} $\mathrm{address\_block}(\mathrm{base}', \mathrm{expt}(2, \mathrm{pe\_size}))(x')$

{1} $\exists\ (\mathrm{pe}: \mathrm{Memory\_Address\_4G}, \mathrm{lvl}: \mathrm{Level}):$
$\mathrm{pe\_in\_pt\_range?}(s', \mathrm{linear\_range}', \mathrm{lvl})(\mathrm{pe}) \wedge$
$\mathrm{address\_block}(\mathrm{pe}, \mathrm{expt}(2, \mathrm{pe\_size}))(x')$

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pe_in_pt_address_in_pt.1`.
`pe_in_pt_address_in_pt.2:`

{-1} $\mathrm{lvl}' < \mathrm{max\_level}$
{-2} $\mathrm{Mem?}(\mathrm{base}'\text{`}\mathrm{type\_of})$
{-3} $0 \leq \mathrm{base}'\text{`}\mathrm{offset}$
{-4} $\mathrm{base}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-5} $\mathrm{pe\_in\_pt\_range?}(s', \mathrm{linear\_range}', \mathrm{lvl}')(\mathrm{base}')$
{-6} $\mathrm{address\_block}(\mathrm{base}', \mathrm{expt}(2, \mathrm{pe\_size}))(x')$

{1} $x'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$

Using lemma pe_in_pt_range_aligned,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of address_block,
Applying disjunctive simplification to flatten sequent,
Using lemma aligned_add_below,
we get 2 subgoals:
`pe_in_pt_address_in_pt.2.1:`

{-1} $\mathrm{pe\_size} \leq \mathrm{bus\_width} \wedge$
$\mathrm{offset}(\mathrm{base}') < \mathrm{expt}(2, \mathrm{bus\_width}) \wedge$
$\mathrm{aligned?}(\mathrm{pe\_size})(\mathrm{offset}(\mathrm{base}')) \wedge \mathrm{expt}(2, \mathrm{pe\_size}) - 1 < \mathrm{expt}(2, \mathrm{pe\_size})$
$\supset \mathrm{offset}(\mathrm{base}') + \mathrm{expt}(2, \mathrm{pe\_size}) - 1 < \mathrm{expt}(2, \mathrm{bus\_width})$
{-2} $\mathrm{pe\_in\_pt\_range?}(s', \mathrm{linear\_range}', \mathrm{lvl}')(\mathrm{base}')$
{-3} $\mathrm{aligned?}(\mathrm{pe\_size})(\mathrm{offset}(\mathrm{base}'))$
{-4} $\mathrm{lvl}' < \mathrm{max\_level}$
{-5} $\mathrm{Mem?}(\mathrm{base}'\text{`}\mathrm{type\_of})$
{-6} $0 \leq \mathrm{base}'\text{`}\mathrm{offset}$
{-7} $\mathrm{base}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-8} $\mathrm{type\_of}(\mathrm{base}') = \mathrm{type\_of}(x')$
{-9} $\mathrm{offset}(x') < \mathrm{offset}(\mathrm{base}') + \mathrm{expt}(2, \mathrm{pe\_size})$

{1} $x'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$

Installing automatic rewrites from: (max_linear_offset! bus_width! pe_size!)
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pe_in_pt_address_in_pt.2.1`.

`pe_in_pt_address_in_pt.2.2`:

{-1}	pe_in_pt_range?(s', linear_range$'$, lvl$'$)(base$'$)
{-2}	aligned?(pe_size)(offset(base$'$))
{-3}	lvl$'$ $<$ max_level
{-4}	Mem?(base$'$`type_of)
{-5}	$0 \leq$ base$'$`offset
{-6}	base$'$`offset $<$ max_linear_offset
{-7}	type_of(base$'$) $=$ type_of(x')
{-8}	offset(x') $<$ offset(base$'$) + expt(2, pe_size)
{1}	expt(2, pe_size) $-$ 1 $\geq$ 0
{2}	x'`offset $<$ max_linear_offset

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pe_in_pt_address_in_pt.2.2`.

`pe_in_pt_address_in_pt.3`:

{-1}	lvl$'$ $<$ max_level
{-2}	Mem?(base$'$`type_of)
{-3}	$0 \leq$ base$'$`offset
{-4}	base$'$`offset $<$ max_linear_offset
{-5}	pe_in_pt_range?(s', linear_range$'$, lvl$'$)(base$'$)
{-6}	address_block(base$'$, expt(2, pe_size))(x')
{1}	$0 \leq$ x'`offset

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `pe_in_pt_address_in_pt.3`.

`pe_in_pt_address_in_pt.4`:

{-1}	lvl$'$ $<$ max_level
{-2}	Mem?(base$'$`type_of)
{-3}	$0 \leq$ base$'$`offset
{-4}	base$'$`offset $<$ max_linear_offset
{-5}	pe_in_pt_range?(s', linear_range$'$, lvl$'$)(base$'$)
{-6}	address_block(base$'$, expt(2, pe_size))(x')
{1}	Mem?(x'`type_of)

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `pe_in_pt_address_in_pt.4`.

Q.E.D.

C.115.11 Linear_Memory_Blessing.is_linear_plain_memory?_TCC1

Terse proof for `is_linear_plain_memory?_TCC1`.

`is_linear_plain_memory?_TCC1:`

{1} ∀ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
(∀ (s: (pm‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))) ∧
pm‘mem = linear_pm
⊃
(∀ (s_1, s_2: (pm‘states)):
OK?[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s_1)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `is_linear_plain_memory?_TCC1`.
Q.E.D.

C.115.12 Linear_Memory_Blessing.is_linear_plain_memory?_TCC2

Terse proof for `is_linear_plain_memory?_TCC2`.
`is_linear_plain_memory?_TCC2:`

{1} ∀ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
(∀ (s: (pm‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))) ∧
pm‘mem = linear_pm
⊃
(∀ (s_1, s_2: (pm‘states)):
OK?[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s_2)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `is_linear_plain_memory?_TCC2`.
Q.E.D.

C.115.13 Linear_Memory_Blessing.is_linear_plain_memory?_TCC3

Terse proof for `is_linear_plain_memory?_TCC3`.
`is_linear_plain_memory?_TCC3:`

{1} ∀ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
(∀ (s: (pm‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))) ∧
(∀ (s_1, s_2: (pm‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2)))
∧
(∀ (s: (pm‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))) ∧
pm‘mem = linear_pm
⊃
(∀ (s_{11}, s_{21}: (pm‘states)):
OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_{11})))

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `is_linear_plain_memory?_TCC3`.

Q.E.D.

C.115.14 Linear_Memory_Blessing.is_linear_plain_memory?_TCC4

Terse proof for `is_linear_plain_memory?_TCC4`.

`is_linear_plain_memory?_TCC4`:

```
{1}  ∀ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
       (∀ (s: (pm`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))) ∧
        (∀ (s1, s2: (pm`states)):
             data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
              data(read_data(pm_phy, segment_reg_data_type)(CS)(s2)))
         ∧
         (∀ (s: (pm`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))) ∧
          pm`mem = linear_pm
        ⊃
        (∀ (s11, s21: (pm`states)):
             OK?[Physical_memory, Pdbr_type]
                  (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s21)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `is_linear_plain_memory?_TCC4`.

Q.E.D.

C.115.15 Linear_Memory_Blessing.is_linear_plain_memory?_TCC5

Terse proof for `is_linear_plain_memory?_TCC5`.

`is_linear_plain_memory?_TCC5:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
       (∀ (s: (pm‘states), lvl: Level,
             a:
               (pe_in_pt_range?(s,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm‘ro_addr ∪ pm‘rw_addr)),
                                lvl))):
          OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s)))
        ∧
        (∀ (s1, s2: (pm‘states)):
           pe_in_pdir_range?(s1,
                             restrict[Address, Memory_Address_4G, boolean]
                               ((pm‘ro_addr ∪ pm‘rw_addr)))
            =
            pe_in_pdir_range?(s2,
                              restrict[Address, Memory_Address_4G, boolean]
                                ((pm‘ro_addr ∪ pm‘rw_addr)))
            ∧
            pe_in_ptab_range?(s1,
                              restrict[Address, Memory_Address_4G, boolean]
                                ((pm‘ro_addr ∪ pm‘rw_addr)))
             =
             pe_in_ptab_range?(s2,
                               restrict[Address, Memory_Address_4G, boolean]
                                 ((pm‘ro_addr ∪ pm‘rw_addr))))
         ∧
         (∀ (s1, s2: (pm‘states)):
            data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
             data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2)))
          ∧
          (∀ (s: (pm‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))) ∧
           (∀ (s1, s2: (pm‘states)):
              data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
               data(read_data(pm_phy, segment_reg_data_type)(CS)(s2)))
            ∧
            (∀ (s: (pm‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)))
             ∧ pm‘mem = linear_pm
        ⊃
        (∀ (s11, s21: (pm‘states), lvl1: Level,
              a1:
                (pe_in_pt_range?(s11,
                                 restrict[Address, Memory_Address_4G, boolean]
                                   ((pm‘ro_addr ∪ pm‘rw_addr)),
                                 lvl1))):
           OK?[Physical_memory, ((range_pt(lvl1)))]
             (read_data[Physical_memory, ((range_pt(lvl1)))]
                (pm_phy, paging_data_type(lvl1))(a1)(s11)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `is_linear_plain_memory?_TCC5`.

Q.E.D.

C.115.16 Linear_Memory_Blessing.is_linear_plain_memory?_TCC6

Terse proof for `is_linear_plain_memory?_TCC6`.

`is_linear_plain_memory?_TCC6:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]]):
       (∀ (s: (pm‘states), lvl: Level,
             a:
               (pe_in_pt_range?(s,
                                restrict[Address, Memory_Address_4G, boolean]
                                  ((pm‘ro_addr ∪ pm‘rw_addr)),
                                lvl))):
          OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s)))
        ∧
        (∀ (s1, s2: (pm‘states)):
           pe_in_pdir_range?(s1,
                             restrict[Address, Memory_Address_4G, boolean]
                               ((pm‘ro_addr ∪ pm‘rw_addr)))
            =
            pe_in_pdir_range?(s2,
                              restrict[Address, Memory_Address_4G, boolean]
                                ((pm‘ro_addr ∪ pm‘rw_addr)))
            ∧
            pe_in_ptab_range?(s1,
                              restrict[Address, Memory_Address_4G, boolean]
                                ((pm‘ro_addr ∪ pm‘rw_addr)))
             =
             pe_in_ptab_range?(s2,
                               restrict[Address, Memory_Address_4G, boolean]
                                 ((pm‘ro_addr ∪ pm‘rw_addr))))
         ∧
         (∀ (s1, s2: (pm‘states)):
            data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
             data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2)))
          ∧
          (∀ (s: (pm‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))) ∧
           (∀ (s1, s2: (pm‘states)):
              data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
               data(read_data(pm_phy, segment_reg_data_type)(CS)(s2)))
            ∧
            (∀ (s: (pm‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)))
             ∧ pm‘mem = linear_pm
        ⊃
        (∀ (s11, s21: (pm‘states), lvl1: Level,
              a1:
                (pe_in_pt_range?(s11,
                                 restrict[Address, Memory_Address_4G, boolean]
                                   ((pm‘ro_addr ∪ pm‘rw_addr)),
                                 lvl1))):
           OK?[Physical_memory, ((range_pt(lvl1)))]
             (read_data[Physical_memory, ((range_pt(lvl1)))]
                (pm_phy, paging_data_type(lvl1))(a1)(s21)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

we get 2 subgoals:

`is_linear_plain_memory?_TCC6.1`:

{-1} $\mathrm{pm}'\text{`states}(s'_{11})$
{-2} $\mathrm{pm}'\text{`states}(s'_{21})$
{-3} $\mathrm{lvl1}' < \mathrm{max\_level}$
{-4} $\mathrm{Mem?}(a'_1\text{`type\_of})$
{-5} $0 \leq a'_1\text{`offset}$
{-6} $a'_1\text{`offset} < \mathrm{max\_linear\_offset}$
{-7} $\mathrm{pe\_in\_pt\_range?}(s'_{11},$
$\quad \mathrm{restrict}[\mathrm{Address}, \mathrm{Memory\_Address\_4G}, \mathrm{boolean}]$
$\quad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})),$
$\quad \mathrm{lvl1}')$
(a'_1)
{-8} $\mathrm{pe\_in\_pdir\_range?}(s'_{11},$
$\quad \mathrm{restrict}[\mathrm{Address}, \mathrm{Memory\_Address\_4G}, \mathrm{boolean}]$
$\quad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
$=$
$\mathrm{pe\_in\_pdir\_range?}(s'_{21},$
$\quad \mathrm{restrict}[\mathrm{Address}, \mathrm{Memory\_Address\_4G}, \mathrm{boolean}]$
$\quad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
$\wedge$
$\mathrm{pe\_in\_ptab\_range?}(s'_{11},$
$\quad \mathrm{restrict}[\mathrm{Address}, \mathrm{Memory\_Address\_4G}, \mathrm{boolean}]$
$\quad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
$=$
$\mathrm{pe\_in\_ptab\_range?}(s'_{21},$
$\quad \mathrm{restrict}[\mathrm{Address}, \mathrm{Memory\_Address\_4G}, \mathrm{boolean}]$
$\quad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
{-9} $\forall\ (s_1, s_2 : (\mathrm{pm}'\text{`states})):$
$\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy}, \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s_1)) =$
$\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy}, \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s_2))$
{-10} $\forall\ (s : (\mathrm{pm}'\text{`states})) : \mathrm{OK?}(\mathrm{read\_data}(\mathrm{pm\_phy}, \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s))$
{-11} $\forall\ (s_1, s_2 : (\mathrm{pm}'\text{`states})):$
$\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy}, \mathrm{segment\_reg\_data\_type})(\mathrm{CS})(s_1)) =$
$\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy}, \mathrm{segment\_reg\_data\_type})(\mathrm{CS})(s_2))$
{-12} $\forall\ (s : (\mathrm{pm}'\text{`states})) : \mathrm{OK?}(\mathrm{read\_data}(\mathrm{pm\_phy}, \mathrm{segment\_reg\_data\_type})(\mathrm{CS})(s))$
{-13} $\mathrm{pm}'\text{`mem} = \mathrm{linear\_pm}$

{1} $\mathrm{pe\_in\_pt\_range?}(s'_{21},$
$\quad \mathrm{restrict}[\mathrm{Address}, \mathrm{Memory\_Address\_4G}, \mathrm{boolean}]$
$\quad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})),$
$\quad \mathrm{lvl1}')$
(a'_1)
{2} $\mathrm{OK?}[\mathrm{Physical\_memory}, ((\mathrm{range\_pt}(\mathrm{lvl1}')))]$
$(\mathrm{read\_data}[\mathrm{Physical\_memory}, ((\mathrm{range\_pt}(\mathrm{lvl1}')))]$
$(\mathrm{pm\_phy}, \mathrm{paging\_data\_type}(\mathrm{lvl1}'))(a'_1)(s'_{21}))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `is_linear_plain_memory?_TCC6.1`.

`is_linear_plain_memory?_TCC6.2:`

```
{-1}  pm'`states(s'11)
{-2}  pm'`states(s'21)
{-3}  lvl1' < max_level
{-4}  Mem?(a'1`type_of)
{-5}  0 ≤ a'1`offset
{-6}  a'1`offset < max_linear_offset
{-7}  pe_in_pt_range?(s'11,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'`ro_addr ∪ pm'`rw_addr)),
                      lvl1')
                     (a'1)
{-8}  ∀ (s: (pm'`states), lvl: Level,
         a:
           (pe_in_pt_range?(s,
                            restrict[Address, Memory_Address_4G, boolean]
                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                            lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-9}  pe_in_pdir_range?(s'11,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'`ro_addr ∪ pm'`rw_addr)))
      =
      pe_in_pdir_range?(s'21,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'`ro_addr ∪ pm'`rw_addr)))
      ∧
      pe_in_ptab_range?(s'11,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'`ro_addr ∪ pm'`rw_addr)))
       =
       pe_in_ptab_range?(s'21,
                         restrict[Address, Memory_Address_4G, boolean]
                           ((pm'`ro_addr ∪ pm'`rw_addr)))
{-10} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} pm'`mem = linear_pm
-------------------------------------------------------------------------
 {1}  pe_in_pt_range?(s'21,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'`ro_addr ∪ pm'`rw_addr)),
                      lvl1')
                     (a'1)
 {2}  OK?[Physical_memory, ((range_pt(lvl1')))]
          (read_data[Physical_memory, ((range_pt(lvl1')))]
                    (pm_phy, paging_data_type(lvl1'))(a'1)(s'21))
```

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `is_linear_plain_memory?_TCC6.2`.
Q.E.D.

C.116 Proofs for Linear_Memory_Blessing_Properties (challenge-linear.pvs)

C.116.1 Linear_Memory_Blessing_Properties.linear_plain_transformers_ok

Terse proof for `linear_plain_transformers_ok`.
`linear_plain_transformers_ok:`

```
{1}  ∀ (pm: Plain_Memory[Physical_memory]):
       is_linear_plain_memory?(pm) ⊃
         transformers_ok?(pm`states,
                          ((memory_read_transformers(pm`mem, (pm`ro_addr ∪ pm`rw_addr)) ∪ memory_write_
```

Installing automatic rewrites from: (linear_resolve_read_transformers_ok linear_resolve_write_transformers_ok linear_resolve_transformer_invariant singleton member)
Repeatedly Skolemizing and flattening,
Rewriting using transformers_ok_all_transformers, matching in *,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of union,
Using lemma pm_states,
Using lemma pm_plain_phy,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`linear_plain_transformers_ok.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  memory_read_transformers(pm′`mem, (pm′`ro_addr ∪ pm′`rw_addr))(q′)
-------------------------------------------------------------------------
{1}   transformers_ok?(pm′`states, singleton(q′))
```

Expanding the definition of memory_read_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -5,
Hiding formulas: -5,
Rewriting using pm_read_linear, matching in *,
Expanding the definition of linear_read,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (pa: Address): FORALL (s: (pm!1`states)): OK?(linear_resolve(a!1, Read)(s)) IMPLIES pa = data(linear_resolve(a!1, Read)(s)),
we get 4 subgoals:

`linear_plain_transformers_ok.1.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-3}  in_memory(min_linear, max_linear)(a′)
{-4}  Mem?(type_of(a′))
{-5}  plain_memory?(pm_phy)
{-6}  pm′‘states = pm_phy‘states
------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states, singleton(expr_2_super(linear_resolve(a′, Read))))
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(linear_resolve(a′, Read)) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_read(pm_phy‘mem)(pa)))))
```

Hiding formulas: 2,
Using lemma linear_resolve_read_transformers_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1) and hiding *,
Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: (s!1 q!2),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Hiding formulas: (-2 2),
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.1.1`.
`linear_plain_transformers_ok.1.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-3}  in_memory(min_linear, max_linear)(a′)
{-4}  Mem?(type_of(a′))
{-5}  plain_memory?(pm_phy)
{-6}  pm′‘states = pm_phy‘states
------------------------------------------------------------------------
{1}   transformer_invariant?(pm′‘states, singleton(expr_2_super(linear_resolve(a′, Read))))
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(linear_resolve(a′, Read)) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_read(pm_phy‘mem)(pa)))))
```

Hiding formulas: 2,
Using lemma linear_resolve_transformer_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1) and hiding *,
Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.1.2`.
`linear_plain_transformers_ok.1.3`:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  union(pm′`ro_addr, pm′`rw_addr)(a′)
{-3}  in_memory(min_linear, max_linear)(a′)
{-4}  Mem?(type_of(a′))
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = pm_phy`states
|-------
{1}   ∀ (s_1: (pm′`states)):
        OK?(linear_resolve(a′, Read)(s_1)) ⊃
         (∀ (s: (pm′`states)):
            OK?(linear_resolve(a′, Read)(s)) ⊃
             data(linear_resolve(a′, Read)(s_1)) = data(linear_resolve(a′, Read)(s)))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(linear_resolve(a′, Read) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_read(pm_phy`mem)(pa)))))
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma linear_resolve_same_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.1.3`.
`linear_plain_transformers_ok.1.4`:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  union(pm′`ro_addr, pm′`rw_addr)(a′)
{-3}  in_memory(min_linear, max_linear)(a′)
{-4}  Mem?(type_of(a′))
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = pm_phy`states
|-------
{1}   ∀ (d:
          (λ (pa: Address):
             ∀ (s: (pm′`states)):
               OK?(linear_resolve(a′, Read)(s)) ⊃
                pa = data(linear_resolve(a′, Read)(s)))):
        transformers_ok?(pm′`states, singleton(expr_2_super(memory_read(pm_phy`mem)(d))))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(linear_resolve(a′, Read) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_read(pm_phy`mem)(pa)))))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: 2,
Using lemma plain_memory_transformers_ok_read_ro_rw,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of transformers_ok?,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Hiding formulas: (-3 -5 2),

Instantiating quantified variables,
Using lemma pm_linear_resolve_read_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -3,
Expanding the definition of memory_read_transformers,
Instantiating quantified variables,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 -4 -5 -6 -7),
Expanding the definition of restrict,
Expanding the definition of subset?,
Expanding the definition of virt_to_phys_range,
Instantiating the top quantifier in -1 with the terms: (d!1),
we get 2 subgoals:
`linear_plain_transformers_ok.1.4.1:`

{-1}	$(d' \in \lambda\ (\text{phy\_a} : \text{Memory\_Address\_4G}) : \exists\ (\text{lin\_a} : ((\lambda\ (s : \text{Memory\_Address\_4G}) : \text{pm}'\text{`ro\_add}$
	$\Rightarrow\ (d' \in \lambda\ (s : \text{Memory\_Address\_4G}) : \text{union}(\text{pm\_phy`ro\_addr}, \text{pm\_phy`rw\_addr})(s))$
{-2}	is_linear_plain_memory?(pm′)
{-3}	OK?(linear_resolve(a', Read)(s'))
{-4}	$x' = \text{expr\_2\_super}(\text{memory\_read}(\text{pm\_phy`mem})(d'))$
{-5}	pm′`states(s')
{-6}	plain_memory?(pm_phy)
{-7}	$d' = \text{data}(\text{linear\_resolve}(a', \text{Read})(s'))$
{-8}	union(pm′`ro_addr, pm′`rw_addr)(a')
{-9}	in_memory(min_linear, max_linear)(a')
{-10}	Mem?(type_of(a'))
{-11}	pm′`states = pm_phy`states
{1}	union(pm_phy`ro_addr, pm_phy`rw_addr)(d')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
we get 3 subgoals:
`linear_plain_transformers_ok.1.4.1.1:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	OK?(linear_resolve(a', Read)(s'))
{-3}	$x' = \text{expr\_2\_super}(\text{memory\_read}(\text{pm\_phy`mem})(d'))$
{-4}	pm′`states(s')
{-5}	plain_memory?(pm_phy)
{-6}	$d' = \text{data}(\text{linear\_resolve}(a', \text{Read})(s'))$
{-7}	union(pm′`ro_addr, pm′`rw_addr)(a')
{-8}	in_memory(min_linear, max_linear)(a')
{-9}	Mem?(type_of(a'))
{-10}	pm′`states = pm_phy`states
{1}	OK?(linear_resolve(a', Read)(s')) $\wedge$ data(linear_resolve(a', Read)(s')) $= d'$
{2}	union(pm_phy`ro_addr, pm_phy`rw_addr)(d')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.1.4.1.1`.

`linear_plain_transformers_ok.1.4.1.2:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  OK?(linear_resolve(a′, Read)(s′))
  {-3}  x′ = expr_2_super(memory_read(pm_phy‘mem)(d′))
  {-4}  pm′‘states(s′)
  {-5}  plain_memory?(pm_phy)
  {-6}  d′ = data(linear_resolve(a′, Read)(s′))
  {-7}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
  {-8}  in_memory(min_linear, max_linear)(a′)
  {-9}  Mem?(type_of(a′))
 {-10}  pm′‘states = pm_phy‘states
--------------------------------------------------------------------------------
  {1}   union[Memory_access]
            (singleton[Memory_access](Read), singleton[Memory_access](Execute))(Read)
  {2}   union(pm_phy‘ro_addr, pm_phy‘rw_addr)(d′)
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.1.4.1.2`.
`linear_plain_transformers_ok.1.4.1.3:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  OK?(linear_resolve(a′, Read)(s′))
  {-3}  x′ = expr_2_super(memory_read(pm_phy‘mem)(d′))
  {-4}  pm′‘states(s′)
  {-5}  plain_memory?(pm_phy)
  {-6}  d′ = data(linear_resolve(a′, Read)(s′))
  {-7}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
  {-8}  in_memory(min_linear, max_linear)(a′)
  {-9}  Mem?(type_of(a′))
 {-10}  pm′‘states = pm_phy‘states
--------------------------------------------------------------------------------
  {1}   union[Memory_Address_4G]
            (λ (s: Memory_Address_4G): pm′‘ro_addr(s),
               λ (s: Memory_Address_4G): pm′‘rw_addr(s))
            (a′)
  {2}   union(pm_phy‘ro_addr, pm_phy‘rw_addr)(d′)
```

Keeping (-7 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.1.4.1.3`.
`linear_plain_transformers_ok.1.4.2:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  OK?(linear_resolve(a′, Read)(s′))
  {-3}  x′ = expr_2_super(memory_read(pm_phy‘mem)(d′))
  {-4}  pm′‘states(s′)
  {-5}  plain_memory?(pm_phy)
  {-6}  d′ = data(linear_resolve(a′, Read)(s′))
  {-7}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
  {-8}  in_memory(min_linear, max_linear)(a′)
  {-9}  Mem?(type_of(a′))
 {-10}  pm′‘states = pm_phy‘states
--------------------------------------------------------------------------------
  {1}   Mem?(d′‘type_of) ∧ 0 ≤ d′‘offset ∧ d′‘offset < max_linear_offset
  {2}   union(pm_phy‘ro_addr, pm_phy‘rw_addr)(d′)
```

Using lemma linear_resolve_memory_address,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.1.4.2`.
`linear_plain_transformers_ok.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  memory_write_transformers(pm′`mem, pm′`rw_addr)(q′)
-------------------------------------------------------------------
{1}   transformers_ok?(pm′`states, singleton(q′))
```

Expanding the definition of memory_write_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -6,
Hiding formulas: -6,
Rewriting using pm_write_linear, matching in *,
Expanding the definition of linear_write,
Using lemma pm_memory_addr,
Installing automatic rewrites from: union_right
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using fexpr_composition_transformers_ok, matching in * where P gets LAMBDA (pa: Address): FORALL (s: (pm!1`states)): OK?(linear_resolve(a!1, Write)(s)) IMPLIES pa = data(linear_resolve(a!1, Write)(s)),
we get 4 subgoals:
`linear_plain_transformers_ok.2.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  in_memory(min_linear, max_linear)(a′)
{-3}  Mem?(type_of(a′))
{-4}  b′ < max_byte
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = pm_phy`states
{-7}  pm′`rw_addr(a′)
-------------------------------------------------------------------
{1}   transformers_ok?(pm′`states, singleton(expr_2_super(linear_resolve(a′, Write))))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(linear_resolve(a′, Write) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_write(pm_phy`mem)(pa, b′)))))
```

Hiding formulas: 2,
Using lemma linear_resolve_write_transformers_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1) and hiding *,
Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.2.1`.

`linear_plain_transformers_ok.2.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  in_memory(min_linear, max_linear)(a′)
{-3}  Mem?(type_of(a′))
{-4}  b′ < max_byte
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = pm_phy`states
{-7}  pm′`rw_addr(a′)
------------------------------------------------------------------------------
{1}   transformer_invariant?(pm′`states, singleton(expr_2_super(linear_resolve(a′, Write))))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(linear_resolve(a′, Write) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_write(pm_phy`mem)(pa, b′)))))
```

Hiding formulas: 2,
Using lemma linear_resolve_transformer_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 1) and hiding *,
Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.2.2`.

`linear_plain_transformers_ok.2.3:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  in_memory(min_linear, max_linear)(a′)
{-3}  Mem?(type_of(a′))
{-4}  b′ < max_byte
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = pm_phy`states
{-7}  pm′`rw_addr(a′)
------------------------------------------------------------------------------
{1}   ∀ (s_1: (pm′`states)):
        OK?(linear_resolve(a′, Write)(s_1)) ⊃
         (∀ (s: (pm′`states)):
            OK?(linear_resolve(a′, Write)(s)) ⊃
             data(linear_resolve(a′, Write)(s_1)) = data(linear_resolve(a′, Write)(s)))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(linear_resolve(a′, Write) ##
                                              (λ (pa: Address):
                                                 mem-
      ory_write(pm_phy`mem)(pa, b′)))))
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma linear_resolve_same_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.2.3`.

`linear_plain_transformers_ok.2.4:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  in_memory(min_linear, max_linear)(a′)
{-3}  Mem?(type_of(a′))
{-4}  b′ < max_byte
{-5}  plain_memory?(pm_phy)
{-6}  pm′‘states = pm_phy‘states
{-7}  pm′‘rw_addr(a′)
------------------------------------------------------------------------
{1}   ∀ (d:
           (λ (pa: Address):
              ∀ (s: (pm′‘states)):
                OK?(linear_resolve(a′, Write)(s)) ⊃
                  pa = data(linear_resolve(a′, Write)(s)))):
        transformers_ok?(pm′‘states,
                         singleton(expr_2_super(memory_write(pm_phy‘mem)(d, b′))))
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(linear_resolve(a′, Write) ##
                                              (λ (pa: Address):
                                                 memory_write(pm_phy‘mem)(pa, b′)))))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: 2,
Using lemma plain_memory_transformers_ok_write_block,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`linear_plain_transformers_ok.2.4.1:`

```
{-1}  plain_memory?(pm_phy)
{-2}  ∀ (s: (pm′‘states)):
        OK?(linear_resolve(a′, Write)(s)) ⊃ d′ = data(linear_resolve(a′, Write)(s))
{-3}  is_linear_plain_memory?(pm′)
{-4}  in_memory(min_linear, max_linear)(a′)
{-5}  Mem?(type_of(a′))
{-6}  b′ < max_byte
{-7}  pm′‘states = pm_phy‘states
{-8}  pm′‘rw_addr(a′)
------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states, memory_write_transformers(pm_phy‘mem, address_block(d′, 1)))
{2}   transformers_ok?(pm′‘states, singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))))
{3}   (address_block(d′, 1) ⊆ pm_phy‘rw_addr)
```

Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Hiding formulas: (1 2),
Instantiating quantified variables,
Expanding the definition of address_block,
Hiding formulas: (-2),
Using lemma pm_linear_resolve_write_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 -3 -5 -6 -7),
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 3 subgoals:
`linear_plain_transformers_ok.2.4.1.1:`

{-1} $(x' \in \text{virt\_to\_phys\_range}(s',\ \text{restrict}\ [\text{Address},\ \text{Memory\_Address\_4G},\ \text{boolean}](\text{pm}'\text{`rw\_addr}),\ \text{singleton(Wr}$
$\Rightarrow (x' \in \text{restrict}[\text{Address},\ \text{Memory\_Address\_4G},\ \text{boolean}](\text{pm\_phy`rw\_addr}))$
{-2} is_linear_plain_memory?(pm′)
{-3} OK?(linear_resolve(a′, Write)(s′))
{-4} pm′`states(s′)
{-5} plain_memory?(pm_phy)
{-6} $(x' \in \lambda\ (a:\ \text{Address}):\ \text{type\_of}(d') = \text{type\_of}(a) \wedge \text{offset}(d') \leq \text{offset}(a) \wedge \text{offset}(a) < 1 + \text{offset}(d'$
{-7} $d' = \text{data}(\text{linear\_resolve}(a', \text{Write})(s'))$
{-8} in_memory(min_linear, max_linear)(a′)
{-9} Mem?(type_of(a′))
{-10} $b' < \text{max\_byte}$
{-11} pm′`states = pm_phy`states
{-12} pm′`rw_addr(a′)

{1} $(x' \in \text{pm\_phy`rw\_addr})$

Expanding the definition of restrict,
Expanding the definition of virt_to_phys_range,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Case splitting on d!1 = x!1,
we get 2 subgoals:
`linear_plain_transformers_ok.2.4.1.1.1:`

{-1} $d' = x'$
{-2} OK?(linear_resolve(a′, Write)(s′))
{-3} is_linear_plain_memory?(pm′)
{-4} pm′`states(s′)
{-5} plain_memory?(pm_phy)
{-6} $\text{type\_of}(d') = \text{type\_of}(x')$
{-7} $\text{offset}(d') \leq \text{offset}(x')$
{-8} $\text{offset}(x') < 1 + \text{offset}(d')$
{-9} $d' = \text{data}(\text{linear\_resolve}(a', \text{Write})(s'))$
{-10} in_memory(min_linear, max_linear)(a′)
{-11} Mem?(type_of(a′))
{-12} $b' < \text{max\_byte}$
{-13} pm′`states = pm_phy`states
{-14} pm′`rw_addr(a′)

{1} $\text{data}(\text{linear\_resolve}(a', \text{Write})(s')) = x'$
{2} pm_phy`rw_addr(x′)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.2.4.1.1.1`.

`linear_plain_transformers_ok.2.4.1.1.2:`

{-1}	OK?(linear_resolve(a', Write)(s'))
{-2}	is_linear_plain_memory?(pm$'$)
{-3}	pm$'$‘states(s')
{-4}	plain_memory?(pm_phy)
{-5}	type_of(d') = type_of(x')
{-6}	offset(d') ≤ offset(x')
{-7}	offset(x') < 1 + offset(d')
{-8}	d' = data(linear_resolve(a', Write)(s'))
{-9}	in_memory(min_linear, max_linear)(a')
{-10}	Mem?(type_of(a'))
{-11}	b' < max_byte
{-12}	pm$'$‘states = pm_phy‘states
{-13}	pm$'$‘rw_addr(a')
{1}	d' = x'
{2}	data(linear_resolve(a', Write)(s')) = x'
{3}	pm_phy‘rw_addr(x')

Keeping (-5 -6 -7 1) and hiding *,

Applying decompose-equality,

This completes the proof of `linear_plain_transformers_ok.2.4.1.1.2`.

`linear_plain_transformers_ok.2.4.1.2:`

{-1}	is_linear_plain_memory?(pm$'$)
{-2}	OK?(linear_resolve(a', Write)(s'))
{-3}	pm$'$‘states(s')
{-4}	plain_memory?(pm_phy)
{-5}	(x' ∈ λ (a : Address) : type_of(d') = type_of(a) ∧ offset(d') ≤ offset(a) ∧ offset(a) < 1
{-6}	d' = data(linear_resolve(a', Write)(s'))
{-7}	in_memory(min_linear, max_linear)(a')
{-8}	Mem?(type_of(a'))
{-9}	b' < max_byte
{-10}	pm$'$‘states = pm_phy‘states
{-11}	pm$'$‘rw_addr(a')
{1}	Mem?(x'‘type_of) ∧ 0 ≤ x'‘offset ∧ x'‘offset < max_linear_offset
{2}	(x' ∈ pm_phy‘rw_addr)

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.2.4.1.2`.

`linear_plain_transformers_ok.2.4.1.3:`

{-1}	$\forall$ (x: Memory_Address_4G): ($x \in$ virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean](pm′rw_addr), singleton(W $\Rightarrow$ ($x \in$ restrict[Address, Memory_Address_4G, boolean](pm_phyrw_addr))
{-2}	is_linear_plain_memory?(pm′)
{-3}	OK?(linear_resolve(a', Write)(s'))
{-4}	pm′states(s')
{-5}	plain_memory?(pm_phy)
{-6}	($x' \in \lambda$ (a: Address): type_of(d') = type_of(a) $\wedge$ offset(d') $\leq$ offset(a) $\wedge$ offset(a) $<$ 1 + offset(d'
{-7}	d' = data(linear_resolve(a', Write)(s'))
{-8}	in_memory(min_linear, max_linear)(a')
{-9}	Mem?(type_of(a'))
{-10}	$b' <$ max_byte
{-11}	pm′states = pm_phystates
{-12}	pm′rw_addr(a')
{1}	Mem?(x'′type_of) $\wedge$ 0 $\leq$ x'′offset $\wedge$ x'′offset $<$ max_linear_offset
{2}	($x' \in$ pm_phyrw_addr)

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.2.4.1.3`.

`linear_plain_transformers_ok.2.4.2:`

{-1}	plain_memory?(pm_phy)
{-2}	transformers_ok?(pm_phystates, memory_write_transformers(pm_phymem, address_block(d', 1)))
{-3}	$\forall$ (s: (pm′states)): OK?(linear_resolve(a', Write)(s)) $\supset$ d' = data(linear_resolve(a', Write)(s))
{-4}	is_linear_plain_memory?(pm′)
{-5}	in_memory(min_linear, max_linear)(a')
{-6}	Mem?(type_of(a'))
{-7}	$b' <$ max_byte
{-8}	pm′states = pm_phystates
{-9}	pm′rw_addr(a')
{1}	(singleton(expr_2_super(memory_write(pm_phymem)(d', b'))) $\subseteq$ memory_write_transformers(pm_phymem, a
{2}	transformers_ok?(pm′states, singleton(expr_2_super(memory_write(pm_phymem)(d', b'))))

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `linear_plain_transformers_ok.2.4.2`.

`linear_plain_transformers_ok.2.4.3:`

{-1}	plain_memory?(pm_phy)
{-2}	$\forall$ (s: (pm′‘states)): OK?(linear_resolve(a', Write)(s)) $\supset$ d' = data(linear_resolve(a', Write)(s))
{-3}	is_linear_plain_memory?(pm′)
{-4}	in_memory(min_linear, max_linear)(a')
{-5}	Mem?(type_of(a'))
{-6}	b' < max_byte
{-7}	pm′‘states = pm_phy‘states
{-8}	pm′‘rw_addr(a')
{1}	(singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))) ⊆ memory_write_transformers(pm_p
{2}	transformers_ok?(pm′‘states, singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))))
{3}	(address_block(d', 1) ⊆ pm_phy‘rw_addr)

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `linear_plain_transformers_ok.2.4.3`.

`linear_plain_transformers_ok.3:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′‘states = pm_phy‘states
{-4}	memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr))(q')
{1}	transformers_ok?(pm′‘states, singleton(q'))

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -6,

Hiding formulas: (-1 -6),

Rewriting using pm_read_side_effect_linear, matching in *,

Expanding the definition of linear_read_side_effect,

Case splitting on null?(bl!1) OR (reg_base(min_linear)(type_of(a!1)) <= a!1 AND a!1 + length(bl!1) <= reg_size(max_linear)(type_of(a!1))),

we get 4 subgoals:

`linear_plain_transformers_ok.3.1:`

{-1} null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
{-2} is_linear_plain_memory?(pm′)
{-3} plain_memory?(pm_phy)
{-4} pm′`states = pm_phy`states
{-5} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))

{1} transformers_ok?(pm′`states,
singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′
∧
a′ + length(bl′) ≤
reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN ap-
ply_side_effects(split
(min_page,
a′,
bl′),
lin-
ear_read_side_effect_in_page)
(s)
ELSE mem-
ory_read_side_effect(pm_phy`mem)
(a′, bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)))

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

`linear_plain_transformers_ok.3.1.1:`

```
{-1}  Mem?(type_of(a'))
{-2}  null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{-3}  is_linear_plain_memory?(pm')
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl') ∨
                                                   (reg_base(min_linear)(type_of(a')) ≤ a'
                                                      ∧
                                                      a' + length(bl') ≤
                                                       reg_size(max_linear)(type_of(a')))
                                                 THEN IF Mem?(type_of(a'))
                                                        THEN ap-
      ply_side_effects(split
                                                                                  (min
                                                                                    a'
                                                                                    bl
                                                                                  lin-
      ear_read_side_effect_in_page)
                                                                                 (s)
                                                           ELSE mem-
      ory_read_side_effect(pm_phy`mem)
                                                                    (a', bl', cp')(s)
                                                           ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on null?(bl!1),
we get 2 subgoals:
`linear_plain_transformers_ok.3.1.1.1:`

```
{-1}  null?(bl')
{-2}  Mem?(type_of(a'))
{-3}  null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{-4}  is_linear_plain_memory?(pm')
{-5}  plain_memory?(pm_phy)
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
------------------------------------------------------------------------
{1}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                apply_side_effects(split(min_page, a', bl'),
                                                                   lin-
      ear_read_side_effect_in_page)
                                                                  (s))))
```

Hiding formulas: -3,
Using lemma split_null,
Expanding the definition of apply_side_effects,
Expanding the definition of reduce,
Expanding the definition of ok_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.3.1.1.1`.
`linear_plain_transformers_ok.3.1.1.2:`

```
{-1}  Mem?(type_of(a'))
{-2}  null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{-3}  is_linear_plain_memory?(pm')
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
------------------------------------------------------------------------
{1}   null?(bl')
{2}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                apply_side_effects(split(min_page, a', bl'),
                                                                   linear_read_side_effect_in_page)
                                                                   (s))))
```

Hiding formulas: -2,
Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -7,
Hiding formulas: -7,
Case splitting on 0 <= a!1`offset AND a!1`offset < max_linear_offset,
we get 2 subgoals:
`linear_plain_transformers_ok.3.1.1.2.1:`

```
{-1}  0 ≤ a'`offset ∧ a'`offset < max_linear_offset
{-2}  Mem?(type_of(a'))
{-3}  is_linear_plain_memory?(pm')
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-7}  pm'`states(s')
------------------------------------------------------------------------
{1}   null?(bl')
{2}   OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         apply_side_effects(split(min_page, a', bl'),
                                            linear_read_side_effect_in_page)
                                           (s))
                       (s'))
```

Using lemma address_block_split_type,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using ok_expr_2_super, matching in *,

Using lemma apply_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_transformers_ok.3.1.1.2.1.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} Mem?(type_of(a′))
{-8} plain_memory?(pm_phy)
{-9} pm′‘states = pm_phy‘states

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
(split(min_page, a′, bl′))
{2} OK?(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
{3} null?(bl′)

Using lemma split_linear_read_side_effects_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.1.2.1.1`.

`linear_plain_transformers_ok.3.1.1.2.1.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} Mem?(type_of(a′))
{-8} plain_memory?(pm_phy)
{-9} pm′‘states = pm_phy‘states

{1} ∀ (s1: (pm′‘states)):
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s1)))
(split(min_page, a′, bl′))
{2} OK?(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
{3} null?(bl′)

Repeatedly Skolemizing and flattening,

Using lemma split_linear_read_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.1.2.1.2`.

`linear_plain_transformers_ok.3.1.1.2.2:`

{-1} Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm$'$)
{-3} plain_memory?(pm_phy)
{-4} pm$'$`states = pm_phy`states
{-5} (address_block(a', length(bl$'$)) $\subseteq$ (pm$'$`ro_addr $\cup$ pm$'$`rw_addr))
{-6} pm$'$`states(s')

{1} $0 \leq a'$`offset $\wedge$ $a'$`offset $<$ max_linear_offset
{2} null?(bl$'$)
{3} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
apply_side_effects(split(min_page, a', bl$'$),
linear_read_side_effect_in_page)
(s))
(s'))

Hiding formulas: 3,

Expanding the definition of length,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of subset?,

Instantiating quantified variables,

Using lemma pm_memory_addr,

Expanding the definition of address_block,

Expanding the definition of in_memory,

Expanding the definition of reg_size,

Expanding the definition of <,

Expanding the definition of max_linear,

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <=,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.1.2.2`.

```
linear_plain_transformers_ok.3.1.2:

{-1}  null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′`states = pm_phy`states
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   Mem?(type_of(a′))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl′) ∨
                                                   (reg_base(min_linear)(type_of(a′)) ≤ a′
                                                      ∧
                                                      a′ + length(bl′) ≤
                                                       reg_size(max_linear)(type_of(a′)))
                                                  THEN IF Mem?(type_of(a′))
                                                         THEN ap-
      ply_side_effects(split
                                                                                  (min
                                                                                    a′
                                                                                    bl
                                                                                  lin-
      ear_read_side_effect_in_page)
                                                                                (s)
                                                                 ELSE mem-
      ory_read_side_effect(pm_phy`mem)
                                                                          (a′, bl′, cp′)(s)
                                                                 ENDIF
                                                  ELSE Fatal
                                                  ENDIF)))
```

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_transformers_ok_read_side_effects_ro_rw,
Using lemma transformers_ok_mono_transformers,
Hiding formulas: -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-2 2),
Expanding the definition of memory_read_side_effect_super_transformers,
Expanding the definition of expr_2_super,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Replacing using formula -1,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -6 with the terms: x'',
Expanding the definition of address_block,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.1.2`.

`linear_plain_transformers_ok.3.2`:

```
{-1}  is_linear_plain_memory?(pm')
{-2}  plain_memory?(pm_phy)
{-3}  pm'`states = pm_phy`states
{-4}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
------------------------------------------------------------------------
{1}   null?(bl') ∨
       (reg_base(min_linear)(type_of(a')) ≤ a' ∧
          a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{2}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl') ∨
                                                    (reg_base(min_linear)(type_of(a')) ≤ a'
                                                       ∧
                                                       a' + length(bl') ≤
                                                         reg_size(max_linear)(type_of(a')))
                                                  THEN IF Mem?(type_of(a'))
                                                         THEN ap-
      ply_side_effects(split
                                                                                (min_page,
                                                                                   a',
                                                                                   bl'),
                                                                                lin-
      ear_read_side_effect_in_page)
                                                                               (s)
                                                       ELSE mem-
      ory_read_side_effect(pm_phy`mem)
                                                                 (a', bl', cp')(s)
                                                       ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Hiding formulas: 2,

Expanding the definition of length,

Applying disjunctive simplification to flatten sequent,

Using lemma pm_memory_addr,

Using lemma pm_memory_addr,

we get 2 subgoals:

`linear_plain_transformers_ok.3.2.1:`

```
{-1}  is_linear_plain_memory?(pm′) ∧
       union(pm′`ro_addr, pm′`rw_addr)(a′ + length(cdr(bl′)))
       ⊃
       in_memory(min_linear, max_linear)(a′ + length(cdr(bl′))) ∧
        Mem?(type_of(a′ + length(cdr(bl′))))
{-2}  is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
       in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-3}  is_linear_plain_memory?(pm′)
{-4}  plain_memory?(pm_phy)
{-5}  pm′`states = pm_phy`states
{-6}  (address_block(a′, CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES) ⊆ (pm′`ro_add
------------------------------------------------------------------------------------------------
{1}   null?(bl′)
{2}   (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
          a′ + CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES ≤
           reg_size(max_linear)(type_of(a′)))
```

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (-4 -3),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.2.1`.

`linear_plain_transformers_ok.3.2.2:`

```
{-1}  is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
       in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′`states = pm_phy`states
{-5}  (address_block(a′, CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES) ⊆ (pm′`ro_add
------------------------------------------------------------------------------------------------
{1}   cons?[Byte](bl′)
{2}   null?(bl′)
{3}   (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
          a′ + CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES ≤
           reg_size(max_linear)(type_of(a′)))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.3.2.2`.

`linear_plain_transformers_ok.3.3:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   ¬ null?(bl′) ∧ reg_base(min_linear)(type_of(a′)) ≤ a′ ⊃ Mem?(max_linear`type_of)
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl′) ∨
                                                   (reg_base(min_linear)(type_of(a′)) ≤ a′
                                                     ∧
                                                     a′ + length(bl′) ≤
                                                       reg_size(max_linear)(type_of(a′)))
                                                THEN IF Mem?(type_of(a′))
                                                       THEN ap-
ply_side_effects(split
                                                                                  (min_page,
                                                                                     a′,
                                                                                     bl′),
                                                                                  lin-
ear_read_side_effect_in_page)
                                                                                 (s)
                                                     ELSE mem-
ory_read_side_effect(pm_phy`mem)
                                                                  (a′, bl′, cp′)(s)
                                                     ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `linear_plain_transformers_ok.3.3`.

`linear_plain_transformers_ok.3.4:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = pm_phy‘states
{-4}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------
{1}   ¬ null?(bl′) ⊃ Mem?(min_linear‘type_of)
{2}   transformers_ok?(pm′‘states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl′) ∨
                                                   (reg_base(min_linear)(type_of(a′)) ≤ a′
                                                      ∧
                                                      a′ + length(bl′) ≤
                                                       reg_size(max_linear)(type_of(a′)))
                                                  THEN IF Mem?(type_of(a′))
                                                         THEN ap-
      ply_side_effects(split
                                                                                  (min
                                                                                    a′
                                                                                    bl
                                                                                  lin-
      ear_read_side_effect_in_page)
                                                                                 (s)
                                                         ELSE mem-
      ory_read_side_effect(pm_phy‘mem)
                                                                  (a′, bl′, cp′)(s)
                                                         ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `linear_plain_transformers_ok.3.4`.

`linear_plain_transformers_ok.4:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′‘states = pm_phy‘states
{-4}  memory_write_side_effect_super_transformers(pm′‘mem, pm′‘rw_addr)(q′)
------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states, singleton(q′))
```

Expanding the definition of memory_write_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -6,

Hiding formulas: (-1 -6),

Rewriting using pm_write_side_effect_linear, matching in *,

Expanding the definition of linear_write_side_effect,

Case splitting on null?(bl!1) OR (reg_base(min_linear)(type_of(a!1)) <= a!1 AND a!1 + length(bl!1) <= reg_size(max_linear)(type_of(a!1))),

we get 4 subgoals:

`linear_plain_transformers_ok.4.1:`

```
{-1}  null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′`states = pm_phy`states
{-5}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
------------------------------------------------------------------------
 {1}  transformers_ok?(pm′`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl′) ∨
                                                     (reg_base(min_linear)(type_of(a′)) ≤ a′
                                                        ∧
                                                        a′ + length(bl′) ≤
                                                          reg_size(max_linear)(type_of(a′)))
                                                  THEN IF Mem?(type_of(a′))
                                                         THEN ap-
      ply_side_effects(split
                                                                                  (min_page,
                                                                                     a′,
                                                                                     bl′),
                                                                                  lin-
      ear_write_side_effect_in_page)
                                                                                 (s)
                                                         ELSE mem-
      ory_write_side_effect(pm_phy`mem)
                                                                    (a′, bl′, cp′)(s)
                                                         ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

`linear_plain_transformers_ok.4.1.1:`

```
{-1}  Mem?(type_of(a'))
{-2}  null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{-3}  is_linear_plain_memory?(pm')
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
{1}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl') ∨
                                                   (reg_base(min_linear)(type_of(a')) ≤ a'
                                                      ∧
                                                      a' + length(bl') ≤
                                                       reg_size(max_linear)(type_of(a')))
                                                  THEN IF Mem?(type_of(a'))
                                                         THEN ap-
      ply_side_effects(split
                                                                                  (min
                                                                                    a'
                                                                                    bl
                                                                                  lin-
      ear_write_side_effect_in_page)
                                                                                (s)
                                                         ELSE mem-
      ory_write_side_effect(pm_phy`mem)
                                                                  (a', bl', cp')(s)
                                                         ENDIF
                                                  ELSE Fatal
                                                  ENDIF)))
```

Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on null?(bl!1),
we get 2 subgoals:
`linear_plain_transformers_ok.4.1.1.1:`

```
{-1}  null?(bl')
{-2}  Mem?(type_of(a'))
{-3}  null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{-4}  is_linear_plain_memory?(pm')
{-5}  plain_memory?(pm_phy)
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
{1}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                apply_side_effects(split(min_page, a', bl'),
                                                                   lin-
      ear_write_side_effect_in_page)
                                                                 (s))))
```

Hiding formulas: -3,
Using lemma split_null,
Expanding the definition of apply_side_effects,
Expanding the definition of reduce,
Expanding the definition of ok_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_transformers_ok.4.1.1.1`.
`linear_plain_transformers_ok.4.1.1.2:`

```
{-1}  Mem?(type_of(a'))
{-2}  null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{-3}  is_linear_plain_memory?(pm')
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
{1}   null?(bl')
{2}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                apply_side_effects(split(min_page, a', bl'),
                                                                   linear_write_side_effect_in_page)
                                                                   (s))))
```

Hiding formulas: -2,
Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -7,
Hiding formulas: -7,
Case splitting on 0 <= a!1`offset AND a!1`offset < max_linear_offset,
we get 2 subgoals:
`linear_plain_transformers_ok.4.1.1.2.1:`

```
{-1}  0 ≤ a'`offset ∧ a'`offset < max_linear_offset
{-2}  Mem?(type_of(a'))
{-3}  is_linear_plain_memory?(pm')
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-7}  pm'`states(s')
------------------------------------------------------------------------
{1}   null?(bl')
{2}   OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         apply_side_effects(split(min_page, a', bl'),
                                            linear_write_side_effect_in_page)
                                           (s))
                       (s'))
```

Using lemma address_block_split_type,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_plain_transformers_ok.4.1.1.2.1.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  every(λ (t: [Address, list[Byte]]):
              Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
        (split(min_page, a′, bl′))
{-3}  0 ≤ a′‘offset
{-4}  a′‘offset < max_linear_offset
{-5}  Mem?(type_of(a′))
{-6}  plain_memory?(pm_phy)
{-7}  pm′‘states = pm_phy‘states
{-8}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9}  pm′‘states(s′)
------------------------------------------------------------------------
{1}   null?(bl′)
{2}   OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         apply_side_effects(split(min_page, a′, bl′),
                                                  linear_write_side_effect_in_page)
                                                (s))
                       (s′))
```

Rewriting using ok_expr_2_super, matching in *,

Using lemma apply_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_transformers_ok.4.1.1.2.1.1.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states(s′)
{-3}  every(λ (t: [Address, list[Byte]]):
              Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
        (split(min_page, a′, bl′))
{-4}  0 ≤ a′‘offset
{-5}  a′‘offset < max_linear_offset
{-6}  Mem?(type_of(a′))
{-7}  plain_memory?(pm_phy)
{-8}  pm′‘states = pm_phy‘states
{-9}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------
{1}   every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm′‘states,
                                     singleton(expr_2_super(linear_write_side_effect_in_page(e))))
        (split(min_page, a′, bl′))
{2}   OK?(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
{3}   null?(bl′)
```

Using lemma split_linear_write_side_effects_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.4.1.1.2.1.1.1`.

`linear_plain_transformers_ok.4.1.1.2.1.1.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′`states(s′)
{-3}  every(λ (t: [Address, list[Byte]]):
                Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
            (split(min_page, a′, bl′))
{-4}  0 ≤ a′`offset
{-5}  a′`offset < max_linear_offset
{-6}  Mem?(type_of(a′))
{-7}  plain_memory?(pm_phy)
{-8}  pm′`states = pm_phy`states
{-9}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
-------------------------------------------------------------------------
{1}   ∀ (s1: (pm′`states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                  OK?(linear_write_side_effect_in_page(e)(s1)))
              (split(min_page, a′, bl′))
{2}   OK?(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
{3}   null?(bl′)
```

Repeatedly Skolemizing and flattening,

Using lemma split_linear_write_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.4.1.1.2.1.1.2`.

`linear_plain_transformers_ok.4.1.1.2.1.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  0 ≤ a′`offset
{-3}  a′`offset < max_linear_offset
{-4}  Mem?(type_of(a′))
{-5}  plain_memory?(pm_phy)
{-6}  pm′`states = pm_phy`states
{-7}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-8}  pm′`states(s′)
-------------------------------------------------------------------------
{1}   (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{2}   null?(bl′)
{3}   OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         apply_side_effects(split(min_page, a′, bl′),
                                            linear_write_side_effect_in_page)
                                           (s))
                       (s′))
```

Hiding formulas: 3,

Keeping (-6 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1`rw_addr,

Rewriting using union_subset3, matching in *,

This completes the proof of `linear_plain_transformers_ok.4.1.1.2.1.2`.

`linear_plain_transformers_ok.4.1.1.2.2:`

{-1} Mem?(type_of(a'))
{-2} is_linear_plain_memory?(pm$'$)
{-3} plain_memory?(pm_phy)
{-4} pm$'$`states = pm_phy`states
{-5} (address_block(a', length(bl$'$)) ⊆ pm$'$`rw_addr)
{-6} pm$'$`states(s')

{1} $0 \leq a'$`offset ∧ $a'$`offset < max_linear_offset
{2} null?(bl$'$)
{3} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
apply_side_effects(split(min_page, a', bl$'$),
linear_write_side_effect_in_page)
(s))
(s'))

Hiding formulas: 3,

Expanding the definition of length,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of subset?,

Instantiating quantified variables,

Using lemma pm_memory_addr,

Expanding the definition of address_block,

Expanding the definition of in_memory,

Expanding the definition of reg_size,

Expanding the definition of <,

Expanding the definition of max_linear,

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <=,

Rewriting using union_right, matching in *,

This completes the proof of `linear_plain_transformers_ok.4.1.1.2.2`.

`linear_plain_transformers_ok.4.1.2:`

```
{-1}  null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′`states = pm_phy`states
{-5}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
------------------------------------------------------------------------
{1}   Mem?(type_of(a′))
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl′) ∨
                                                   (reg_base(min_linear)(type_of(a′)) ≤ a′
                                                     ∧
                                                     a′ + length(bl′) ≤
                                                      reg_size(max_linear)(type_of(a′)))
                                                THEN IF Mem?(type_of(a′))
                                                       THEN ap-
      ply_side_effects(split
                                                                                  (min_page,
                                                                                    a′,
                                                                                    bl′),
                                                                                  lin-
      ear_write_side_effect_in_page)
                                                                                 (s)
                                                       ELSE mem-
      ory_write_side_effect(pm_phy`mem)
                                                                 (a′, bl′, cp′)(s)
                                                       ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_transformers_ok_write_side_effects_ro_rw,
Using lemma transformers_ok_mono_transformers,
Hiding formulas: -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-2 2),
Expanding the definition of memory_write_side_effect_super_transformers,
Expanding the definition of expr_2_super,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Replacing using formula -1,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -6 with the terms: x'',
Expanding the definition of address_block,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using union_right, matching in *,

This completes the proof of `linear_plain_transformers_ok.4.1.2`.

`linear_plain_transformers_ok.4.2`:

```
{-1}  is_linear_plain_memory?(pm')
{-2}  plain_memory?(pm_phy)
{-3}  pm'`states = pm_phy`states
{-4}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
{1}   null?(bl') ∨
        (reg_base(min_linear)(type_of(a')) ≤ a' ∧
           a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
{2}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl') ∨
                                                     (reg_base(min_linear)(type_of(a')) ≤ a'
                                                        ∧
                                                        a' + length(bl') ≤
                                                         reg_size(max_linear)(type_of(a')))
                                                  THEN IF Mem?(type_of(a'))
                                                         THEN ap-
      ply_side_effects(split
                                                                                  (min
                                                                                    a'
                                                                                    bl
                                                                                  lin-
      ear_write_side_effect_in_page)
                                                                                 (s)
                                                         ELSE mem-
      ory_write_side_effect(pm_phy`mem)
                                                                  (a', bl', cp')(s)
                                                         ENDIF
                                                  ELSE Fatal
                                                  ENDIF)))
```

Hiding formulas: 2,

Expanding the definition of length,

Using lemma pm_memory_addr,

Using lemma pm_memory_addr,

we get 2 subgoals:

`linear_plain_transformers_ok.4.2.1:`

{-1} is_linear_plain_memory?(pm′) ∧
union(pm′`ro_addr, pm′`rw_addr)(a′ + length(cdr(bl′)))
⊃
in_memory(min_linear, max_linear)(a′ + length(cdr(bl′))) ∧
Mem?(type_of(a′ + length(cdr(bl′))))
{-2} is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-3} is_linear_plain_memory?(pm′)
{-4} plain_memory?(pm_phy)
{-5} pm′`states = pm_phy`states
{-6} (address_block(a′, CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES) ⊆ pm′`rw_addr)

{1} null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES ≤
reg_size(max_linear)(type_of(a′)))

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (-4 -3),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformers_ok.4.2.1`.

`linear_plain_transformers_ok.4.2.2:`

{-1} is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-2} is_linear_plain_memory?(pm′)
{-3} plain_memory?(pm_phy)
{-4} pm′`states = pm_phy`states
{-5} (address_block(a′, CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES) ⊆ pm′`rw_addr)

{1} cons?⌈Byte⌋(bl′)
{2} null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + CASES bl′ OF null : 0, cons(x, y) : length(y) + 1 ENDCASES ≤
reg_size(max_linear)(type_of(a′)))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_plain_transformers_ok.4.2.2`.

`linear_plain_transformers_ok.4.3:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  plain_memory?(pm_phy)
{-3}  pm'`states = pm_phy`states
{-4}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
------------------------------------------------------------------------
{1}   ¬ null?(bl') ∧ reg_base(min_linear)(type_of(a')) ≤ a' ⊃ Mem?(max_linear`type_of)
{2}   transformers_ok?(pm'`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl') ∨
                                                   (reg_base(min_linear)(type_of(a')) ≤ a'
                                                      ∧
                                                      a' + length(bl') ≤
                                                       reg_size(max_linear)(type_of(a')))
                                                  THEN IF Mem?(type_of(a'))
                                                         THEN ap-
      ply_side_effects(split
                                                                                  (min
                                                                                    a'
                                                                                    bl
                                                                                  lin-
      ear_write_side_effect_in_page)
                                                                                (s)
                                                              ELSE mem-
      ory_write_side_effect(pm_phy`mem)
                                                                        (a', bl', cp')(s)
                                                              ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `linear_plain_transformers_ok.4.3`.

linear_plain_transformers_ok.4.4:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
------------------------------------------------------------------------
{1}   ¬ null?(bl′) ⊃ Mem?(min_linear`type_of)
{2}   transformers_ok?(pm′`states,
                       singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                IF null?(bl′) ∨
                                                   (reg_base(min_linear)(type_of(a′)) ≤ a′
                                                      ∧
                                                      a′ + length(bl′) ≤
                                                       reg_size(max_linear)(type_of(a′)))
                                                  THEN IF Mem?(type_of(a′))
                                                         THEN ap-
      ply_side_effects(split
                                                                                (min_page,
                                                                                   a′,
                                                                                   bl′),
                                                                                lin-
      ear_write_side_effect_in_page)
                                                                              (s)
                                                       ELSE mem-
      ory_write_side_effect(pm_phy`mem)
                                                                    (a′, bl′, cp′)(s)
                                                       ENDIF
                                                ELSE Fatal
                                                ENDIF)))
```

Expanding the definition of min_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of linear_plain_transformers_ok.4.4.
Q.E.D.

C.116.2 Linear_Memory_Blessing_Properties.linear_plain_unchanged_memory_invariant

Terse proof for linear_plain_unchanged_memory_invariant.
linear_plain_unchanged_memory_invariant:

```
------------------------------------------------------------------------
{1}   ∀ (pm: Plain_Memory[Physical_memory]):
        is_linear_plain_memory?(pm) ⊃
          unchanged_memory_invariant?(pm`mem, pm`states,
                                      ((pm`other_actions ∪ memory_read_transformers(pm`mem, (pm`ro_addr ∪
                                      (pm`ro_addr ∪ pm`rw_addr))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: has_next_state singleton subset_reflexive pm_write_linear linear_write Mem in_memory Mem reg_size union_left max_linear <
Using lemma linear_unchanged_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:
`linear_plain_unchanged_memory_invariant.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))

{1} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
(pm′‘ro_addr ∪ pm′‘rw_addr))

Case splitting on extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr))) = union(pm!1‘ro_addr, pm!1‘rw_addr),
we get 2 subgoals:
`linear_plain_unchanged_memory_invariant.1.1:`

{-1} extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
= (pm′‘ro_addr ∪ pm′‘rw_addr)
{-2} is_linear_plain_memory?(pm′)
{-3} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))

{1} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
(pm′‘ro_addr ∪ pm′‘rw_addr))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.1.1`.
`linear_plain_unchanged_memory_invariant.1.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))

{1} extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
= (pm′‘ro_addr ∪ pm′‘rw_addr)
{2} unchanged_memory_invariant?(pm′‘mem, pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm
(pm′‘ro_addr ∪ pm′‘rw_addr))

Hiding formulas: (-2 2),
Expanding the definition of extend,
Expanding the definition of restrict,

Applying decompose-equality,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.1.2`.

`linear_plain_unchanged_memory_invariant.2`:

```
{-1}  is_linear_plain_memory?(pm')
{1}   ∀ (s: (pm'`states)):
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    ((pm'`other_actions ∪ memory_read_transformers(pm'`mem, (pm'`ro_add
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s,
                                                          restrict
                                                            [Address, Mem-
ory_Address_4G, boolean]
                                                            ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  ((pm'`other_actions ∪ memory_read_transformers(pm'`mem, (pm'`ro_addr ∪
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using unchanged_memory_invariant_all_transformers, matching in * where pm gets pm_phy`mem,

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Expanding the definition of member,

Using lemma pm_plain_phy,

Using lemma pm_states,

Using lemma plain_memory_unchanged_invariant,

Case splitting on subset?(extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s!1, restrict [Address, Memory_Address_4G, boolean] (union(pm!1`ro_addr, pm!1`rw_addr)), union (singleton(Read), singleton(Execute)))), union(pm_phy`ro_addr, pm_phy`rw_addr)),

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1:`

{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-2} plain_memory?(pm_phy) ⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-3} is_linear_plain_memory?(pm') ⊃ pm'‘states = pm_phy‘states
{-4} is_linear_plain_memory?(pm') ⊃ plain_memory?(pm_phy)
{-5} (pm'‘other_actions(q')) ∨
memory_read_transformers(pm'‘mem, (pm'‘ro_addr ∪ pm'‘rw_addr))(q'))
∨
memory_read_side_effect_super_transformers(pm'‘mem, (pm'‘ro_addr ∪ pm'‘rw_addr))(q')
∨ memory_write_side_effect_super_transformers(pm'‘mem, pm'‘rw_addr)(q')
{-6} pm'‘states(s')
{-7} is_linear_plain_memory?(pm')

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(q'),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr
(singleton(Read) ∪ singleton(Exec

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`linear_plain_unchanged_memory_invariant.2.1.1:`

{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-2} plain_memory?(pm_phy)
{-3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘me
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-4} is_linear_plain_memory?(pm')
{-5} pm'‘states = pm_phy‘states
{-6} pm'‘other_actions(q')
{-7} pm'‘states(s')

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(q'),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr
(singleton(Read) ∪ singleton(Exec

Hiding formulas: (-1 2),
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-11 -16 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.2`:

```
{-1}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-2}  plain_memory?(pm_phy)
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'`states = pm_phy`states
{-6}  memory_read_transformers(pm'`mem, (pm'`ro_addr ∪ pm'`rw_addr))(q')
{-7}  pm'`states(s')
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(q'),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
ory_Address_4G, boolean]
                                                                  ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Expanding the definition of memory_read_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -7,

Hiding formulas: -7,

Rewriting using pm_read_linear, matching in *,

Expanding the definition of linear_read,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using plain_memory_unchanged_composition, matching in * where P gets LAMBDA (pa: Address): FORALL (s: (pm_phy`states)): OK?(linear_resolve(a!1, Read)(s)) IMPLIES pa = data(linear_resolve(a!1, Read)(s)),

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.2.1:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-3}  reg_base(min_linear)(a'`type_of) ≤ a'
{-4}  a'`type_of = a'`type_of
{-5}  a'`offset < max_linear_offset
{-6}  Mem?(type_of(a'))
{-7}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-8}  plain_memory?(pm_phy)
{-9}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-10} pm'`states = pm_phy`states
{-11} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (s_1: (pm_phy`states)):
        OK?(linear_resolve(a', Read)(s_1)) ⊃
         (∀ (s: (pm_phy`states)):
              OK?(linear_resolve(a', Read)(s)) ⊃
               data(linear_resolve(a', Read)(s_1)) = data(linear_resolve(a', Read)(s)))
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(a', Read) ##
                                                         (λ (pa: Address):
                                                            mem-
      ory_read(pm_phy`mem)(pa)))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s',
                                                            restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_add
                                                            (singleton(Read) ∪ singleton(Ex
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma linear_resolve_same_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.1`.

`linear_plain_unchanged_memory_invariant.2.1.2.2:`

```
  {-1}  is_linear_plain_memory?(pm')
  {-2}  union(pm'`ro_addr, pm'`rw_addr)(a')
  {-3}  reg_base(min_linear)(a'`type_of) ≤ a'
  {-4}  a'`type_of = a'`type_of
  {-5}  a'`offset < max_linear_offset
  {-6}  Mem?(type_of(a'))
  {-7}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
  {-8}  plain_memory?(pm_phy)
  {-9}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                    (pm_phy`ro_addr ∪ pm_phy`rw_addr))
 {-10}  pm'`states = pm_phy`states
 {-11}  pm'`states(s')
-------------------------------------------------------------------------------
  {1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(linear_resolve(a', Read))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s',
                                                            restrict
                                                                [Address, Mem-
ory_Address_4G, boolean]
                                                                ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                            (singleton(Read) ∪ singleton(Execute)))))
  {2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(linear_resolve(a', Read) ##
                                                           (λ (pa: Address):
                                                               mem-
ory_read(pm_phy`mem)(pa)))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s',
                                                            restrict
                                                                [Address, Mem-
ory_Address_4G, boolean]
                                                                ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                            (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: 2,

Using lemma linear_resolve_unchanged_pm_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset? member extend

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Rewriting using pm_resolve_address, matching in *,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.2.2.1:`

```
{-1}   extend[Address, Memory_Address_4G, bool, FALSE]
           (virt_to_phys_range(s',
                               restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'`ro_addr ∪ pm'`rw_addr)),
                               (singleton(Read) ∪ singleton(Execute))))
           (x')
{-2}   union(pm'`ro_addr, pm'`rw_addr)(a')
{-3}   is_linear_plain_memory?(pm')
{-4}   pm'`states(s')
{-5}   unchanged_memory_invariant?[Physical_memory]
           (pm_phy`mem, pm'`states,
            singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))),
            ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ extend [Address, Memory_Address_4G, bool, FA
{-6}   reg_base(min_linear)(a'`type_of) ≤ a'
{-7}   a'`type_of = a'`type_of
{-8}   a'`offset < max_linear_offset
{-9}   Mem?(type_of(a'))
{-10}  ∀ (x: Address):
         extend[Address, Memory_Address_4G, bool, FALSE]
             (virt_to_phys_range(s',
                                 restrict[Address, Memory_Address_4G, boolean]
                                     ((pm'`ro_addr ∪ pm'`rw_addr)),
                                 (singleton(Read) ∪ singleton(Execute))))
             (x)
           ⇒ union(pm_phy`ro_addr, pm_phy`rw_addr)(x)
{-11}  plain_memory?(pm_phy)
{-12}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                   ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                   (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-13}  pm'`states = pm_phy`states
-------------------------------------------------------------------------------
 {1}   virt_to_phys_range(s',
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                          (singleton(Read) ∪ singleton(Execute)))
                         (x')
 {2}   difference((pm_phy`ro_addr ∪ pm_phy`rw_addr),
                  extend[Address, Memory_Address_4G, bool, FALSE]
                      (address_in_pt_range?(s',
                                            restrict[Address, Mem-
       ory_Address_4G, boolean]
                                                ((pm'`ro_addr ∪ pm'`rw_addr)))))
                  (x')
 {3}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                   singleton(expr_2_super(linear_resolve(a', Read))),
                                   extend[Address, Memory_Address_4G, bool, FALSE]
                                       (virt_to_phys_range(s',
                                                           restrict
                                                               [Address, Mem-
       ory_Address_4G, boolean]
                                                               ((pm'`ro_addr ∪ pm'`rw_add
                                                           (singleton(Read) ∪ singleton(Ex
```

Expanding the definition of extend,

which is trivially true.

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.2.1`.

`linear_plain_unchanged_memory_invariant.2.1.2.2.2`:

```
{-1}  extend[Address, Memory_Address_4G, bool, FALSE]
        (virt_to_phys_range(s',
                            restrict[Address, Memory_Address_4G, boolean]
                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                            (singleton(Read) ∪ singleton(Execute))))
        (x')
{-2}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-3}  is_linear_plain_memory?(pm')
{-4}  pm'`states(s')
{-5}  unchanged_memory_invariant?[Physical_memory]
        (pm_phy`mem, pm'`states,
          singleton(expr_2_super[Physical_memory, Address](linear_resolve(a', Read))),
          ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](addres
{-6}  reg_base(min_linear)(a'`type_of) ≤ a'
{-7}  a'`type_of = a'`type_of
{-8}  a'`offset < max_linear_offset
{-9}  Mem?(type_of(a'))
{-10} ∀ (x: Address):
        extend[Address, Memory_Address_4G, bool, FALSE]
          (virt_to_phys_range(s',
                              restrict[Address, Memory_Address_4G, boolean]
                                ((pm'`ro_addr ∪ pm'`rw_addr)),
                              (singleton(Read) ∪ singleton(Execute))))
          (x)
        ⇒ union(pm_phy`ro_addr, pm_phy`rw_addr)(x)
{-11} plain_memory?(pm_phy)
{-12} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-13} pm'`states = pm_phy`states
-------------------------------------------------------------------------------
 {1}  Mem?(x'`type_of) ∧ 0 ≤ x'`offset ∧ x'`offset < max_linear_offset
 {2}  difference((pm_phy`ro_addr ∪ pm_phy`rw_addr),
                 extend[Address, Memory_Address_4G, bool, FALSE]
                   (address_in_pt_range?(s',
                                         restrict[Address, Mem-
ory_Address_4G, boolean]
                                           ((pm'`ro_addr ∪ pm'`rw_addr)))))
                 (x')
 {3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(a', Read))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s',
                                                        restrict
                                                          [Address, Mem-
ory_Address_4G, boolean]
                                                          ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                        (singleton(Read) ∪ singleton(Execute)))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.2.2`.
`linear_plain_unchanged_memory_invariant.2.1.2.3`:

```
{-1}  is_linear_plain_memory?(pm')
{-2}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-3}  reg_base(min_linear)(a'`type_of) ≤ a'
{-4}  a'`type_of = a'`type_of
{-5}  a'`offset < max_linear_offset
{-6}  Mem?(type_of(a'))
{-7}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-8}  plain_memory?(pm_phy)
{-9}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-10} pm'`states = pm_phy`states
{-11} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (d:
          (λ (pa: Address):
             ∀ (s: (pm_phy`states)):
               OK?(linear_resolve(a', Read)(s)) ⊃
                pa = data(linear_resolve(a', Read)(s)))):
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_read(pm_phy`mem)(d))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                            [Address, Mem-
      ory_Address_4G, boolean]
                                                            ((pm'`ro_addr ∪ pm'`rw_a
                                                          (singleton(Read) ∪ singleton(E
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(a', Read) ##
                                                         (λ (pa: Address):
                                                            mem-
      ory_read(pm_phy`mem)(pa)))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s',
                                                        restrict
                                                          [Address, Mem-
      ory_Address_4G, boolean]
                                                          ((pm'`ro_addr ∪ pm'`rw_add
                                                        (singleton(Read) ∪ singleton(Ex
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Using lemma memory_read_transformers_memory_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.2.3.1:`

```
{-1}  memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr))
                                    (expr_2_super(memory_read(pm_phy`mem)(d′)))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                    (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4}  ∀ (s: (pm_phy`states)):
        OK?(linear_resolve(a′, Read)(s)) ⊃ d′ = data(linear_resolve(a′, Read)(s))
{-5}  is_linear_plain_memory?(pm′)
{-6}  union(pm′`ro_addr, pm′`rw_addr)(a′)
{-7}  reg_base(min_linear)(a′`type_of) ≤ a′
{-8}  a′`type_of = a′`type_of
{-9}  a′`offset < max_linear_offset
{-10} Mem?(type_of(a′))
{-11} plain_memory?(pm_phy)
{-12} pm′`states = pm_phy`states
{-13} pm′`states(s′)
------------------------------------------------------------------------------
 {1}  (singleton(expr_2_super(memory_read(pm_phy`mem)(d′))) ⊆ ((pm_phy`other_actions ∪ memory_read_transfor
```

Keeping (-1 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.3.1`.
`linear_plain_unchanged_memory_invariant.2.1.2.3.2:`

```
{-1}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
{-2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                    (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-3}  ∀ (s: (pm_phy`states)):
        OK?(linear_resolve(a′, Read)(s)) ⊃ d′ = data(linear_resolve(a′, Read)(s))
{-4}  is_linear_plain_memory?(pm′)
{-5}  union(pm′`ro_addr, pm′`rw_addr)(a′)
{-6}  reg_base(min_linear)(a′`type_of) ≤ a′
{-7}  a′`type_of = a′`type_of
{-8}  a′`offset < max_linear_offset
{-9}  Mem?(type_of(a′))
{-10} plain_memory?(pm_phy)
{-11} pm′`states = pm_phy`states
{-12} pm′`states(s′)
------------------------------------------------------------------------------
 {1}  union(pm_phy`ro_addr, pm_phy`rw_addr)(d′)
 {2}  (singleton(expr_2_super(memory_read(pm_phy`mem)(d′))) ⊆ ((pm_phy`other_actions ∪ memory_read_transfor
```

Instantiating quantified variables,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Using lemma pm_linear_resolve_read_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-3 -4 -6 -7 -8 -9),
Expanding the definition of subset?,
Instantiating quantified variables,
we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.1:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	OK?(linear_resolve(a′, Read)(s′))
{-3}	(d′ ∈ virt_to_phys_range(s′, (restrict [Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ r ⇒ (d′ ∈ restrict[Address, Memory_Address_4G, boolean]((pm_phy`ro_addr ∪ pm_phy`rw_addr)))
{-4}	∀ (x: Address): (x ∈ extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [A ⇒ (x ∈ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}	unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-6}	d′ = data(linear_resolve(a′, Read)(s′))
{-7}	union(pm′`ro_addr, pm′`rw_addr)(a′)
{-8}	reg_base(min_linear)(a′`type_of) ≤ a′
{-9}	a′`type_of = a′`type_of
{-10}	a′`offset < max_linear_offset
{-11}	Mem?(type_of(a′))
{-12}	plain_memory?(pm_phy)
{-13}	pm′`states = pm_phy`states
{-14}	pm′`states(s′)
{1}	union(pm_phy`ro_addr, pm_phy`rw_addr)(d′)
{2}	∀ (x: [Physical_memory → SuperResult[Physical_memory]]): (x ∈ singleton(expr_2_super(memory_read(pm_phy`mem)(d′)))) ⇒ (x ∈ ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ p

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  OK?(linear_resolve(a′, Read)(s′))
{-3}  restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(d′)
{-4}  ∀ (x: Address):
        extend[Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range(s′,
                                restrict[Address, Memory_Address_4G, boolean]
                                    ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                                (singleton(Read) ∪ singleton(Execute))))
            (x)
          ⇒ union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x)
{-5}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6}  d′ = data(linear_resolve(a′, Read)(s′))
{-7}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-8}  reg_base(min_linear)(a′‘type_of) ≤ a′
{-9}  a′‘type_of = a′‘type_of
{-10} a′‘offset < max_linear_offset
{-11} Mem?(type_of(a′))
{-12} plain_memory?(pm_phy)
{-13} pm′‘states = pm_phy‘states
{-14} pm′‘states(s′)
------------------------------------------------------------------------
 {1}  union(pm_phy‘ro_addr, pm_phy‘rw_addr)(d′)
 {2}  ∀ (x: [Physical_memory → SuperResult[Physical_memory]]):
        x = expr_2_super(memory_read(pm_phy‘mem)(d′)) ⇒
          union((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_
                (memory_read_side_effect_super_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr
            (x)
```

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  OK?(linear_resolve(a', Read)(s'))
{-3}  ∀ (x: Address):
        extend[Address, Memory_Address_4G, bool, FALSE]
            (virt_to_phys_range(s',
                                restrict[Address, Memory_Address_4G, boolean]
                                    ((pm'`ro_addr ∪ pm'`rw_addr)),
                                (singleton(Read) ∪ singleton(Execute))))
            (x)
          ⇒ union(pm_phy`ro_addr, pm_phy`rw_addr)(x)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  d' = data(linear_resolve(a', Read)(s'))
{-6}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-7}  reg_base(min_linear)(a'`type_of) ≤ a'
{-8}  a'`type_of = a'`type_of
{-9}  a'`offset < max_linear_offset
{-10} Mem?(type_of(a'))
{-11} plain_memory?(pm_phy)
{-12} pm'`states = pm_phy`states
{-13} pm'`states(s')
------------------------------------------------------------------------
 {1}  virt_to_phys_range(s',
                         (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr) ∪ restrict
                         (singleton(Read) ∪ singleton(Execute)))
                        (d')
 {2}  union(pm_phy`ro_addr, pm_phy`rw_addr)(d')
 {3}  ∀ (x: [Physical_memory → SuperResult[Physical_memory]]):
        x = expr_2_super(memory_read(pm_phy`mem)(d')) ⇒
          union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ p
                (memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_p
              (x)
```

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  OK?(linear_resolve(a′, Read)(s′))
{-3}  ∀ (x: Address):
        extend[Address, Memory_Address_4G, bool, FALSE]
          (λ (phy_a: Memory_Address_4G):
             ∃ (lin_a:
                  (restrict[Address, Memory_Address_4G, boolean]
                     ((pm′`ro_addr ∪ pm′`rw_addr))),
                ac: ((singleton(Read) ∪ singleton(Execute)))):
               OK?(linear_resolve(lin_a, ac)(s′)) ∧
                data(linear_resolve(lin_a, ac)(s′)) = phy_a)
          (x)
        ⇒ union(pm_phy`ro_addr, pm_phy`rw_addr)(x)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  d′ = data(linear_resolve(a′, Read)(s′))
{-6}  union(pm′`ro_addr, pm′`rw_addr)(a′)
{-7}  reg_base(min_linear)(a′`type_of) ≤ a′
{-8}  a′`type_of = a′`type_of
{-9}  a′`offset < max_linear_offset
{-10} Mem?(type_of(a′))
{-11} plain_memory?(pm_phy)
{-12} pm′`states = pm_phy`states
{-13} pm′`states(s′)
-------------------------------------------------------------------------------
 {1}  OK?(linear_resolve(a′, Read)(s′)) ∧ data(linear_resolve(a′, Read)(s′)) = d′
 {2}  union(pm_phy`ro_addr, pm_phy`rw_addr)(d′)
 {3}  ∀ (x: [Physical_memory → SuperResult[Physical_memory]]):
        x = expr_2_super(memory_read(pm_phy`mem)(d′)) ⇒
          union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_
                  (memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)
                (x)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.1`.

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.2:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  OK?(linear_resolve(a', Read)(s'))
{-3}  ∀ (x: Address):
        extend[Address, Memory_Address_4G, bool, FALSE]
          (λ (phy_a: Memory_Address_4G):
             ∃ (lin_a:
                  (restrict[Address, Memory_Address_4G, boolean]
                     ((pm'`ro_addr ∪ pm'`rw_addr))),
                ac: ((singleton(Read) ∪ singleton(Execute)))):
               OK?(linear_resolve(lin_a, ac)(s')) ∧
                data(linear_resolve(lin_a, ac)(s')) = phy_a)
          (x)
        ⇒ union(pm_phy`ro_addr, pm_phy`rw_addr)(x)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  d' = data(linear_resolve(a', Read)(s'))
{-6}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-7}  reg_base(min_linear)(a'`type_of) ≤ a'
{-8}  a'`type_of = a'`type_of
{-9}  a'`offset < max_linear_offset
{-10} Mem?(type_of(a'))
{-11} plain_memory?(pm_phy)
{-12} pm'`states = pm_phy`states
{-13} pm'`states(s')
------------------------------------------------------------------------
 {1}  union[Memory_Address_4G]
        (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
          restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
        (a')
 {2}  union(pm_phy`ro_addr, pm_phy`rw_addr)(d')
 {3}  ∀ (x: [Physical_memory → SuperResult[Physical_memory]]):
        x = expr_2_super(memory_read(pm_phy`mem)(d')) ⇒
         union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem,  (pm_phy`ro_addr ∪ p
                 (memory_read_side_effect_super_transformers(pm_phy`mem,  (pm_phy`ro_addr ∪ pm_p
           (x)
```

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.3.2.1.2.2`.

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} OK?(linear_resolve(a′, Read)(s′))
{-3} ∀ (x: Address):
(x ∈ extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Me
⇒ (x ∈ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-5} d′ = data(linear_resolve(a′, Read)(s′))
{-6} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-7} reg_base(min_linear)(a′‘type_of) ≤ a′
{-8} a′‘type_of = a′‘type_of
{-9} a′‘offset < max_linear_offset
{-10} Mem?(type_of(a′))
{-11} plain_memory?(pm_phy)
{-12} pm′‘states = pm_phy‘states
{-13} pm′‘states(s′)

{1} Mem?(d′‘type_of) ∧ 0 ≤ d′‘offset ∧ d′‘offset < max_linear_offset
{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(d′)
{3} ∀ (x: [Physical_memory → SuperResult[Physical_memory]]):
(x ∈ singleton(expr_2_super(memory_read(pm_phy‘mem)(d′)))) ⇒
(x ∈ ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_a

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.3.2.2`.

`linear_plain_unchanged_memory_invariant.2.1.2.3.2.3:`

{-1} is_linear_plain_memory?(pm′)
{-2} OK?(linear_resolve(a′, Read)(s′))
{-3} ∀ (x: Memory_Address_4G):
(x ∈ virt_to_phys_range(s′, (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪
⇒
(x ∈ restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
{-4} ∀ (x: Address):
(x ∈ extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [A
⇒ (x ∈ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6} d′ = data(linear_resolve(a′, Read)(s′))
{-7} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-8} reg_base(min_linear)(a′‘type_of) ≤ a′
{-9} a′‘type_of = a′‘type_of
{-10} a′‘offset < max_linear_offset
{-11} Mem?(type_of(a′))
{-12} plain_memory?(pm_phy)
{-13} pm′‘states = pm_phy‘states
{-14} pm′‘states(s′)

{1} Mem?(d′‘type_of) ∧ 0 ≤ d′‘offset ∧ d′‘offset < max_linear_offset
{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(d′)
{3} ∀ (x: [Physical_memory → SuperResult[Physical_memory]]):
(x ∈ singleton(expr_2_super(memory_read(pm_phy‘mem)(d′)))) ⇒
(x ∈ ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ p

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Using lemma pm_linear_resolve_read_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.2.3.2.3`.

`linear_plain_unchanged_memory_invariant.2.1.3:`

{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-2} plain_memory?(pm_phy)
{-3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘me
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states = pm_phy‘states
{-6} memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr))(q′)
{-7} pm′‘states(s′)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(q′),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr
(singleton(Read) ∪ singleton(Exe

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Hiding formulas: (-1 -8),

Rewriting using pm_read_side_effect_linear, matching in *,

Expanding the definition of linear_read_side_effect,

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1:`

{-1} Mem?(type_of(a'))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5} is_linear_plain_memory?(pm′)
{-6} pm′`states = pm_phy`states
{-7} (address_block(a', length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-8} pm′`states(s')

{1} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
IF null?(bl′) ∨
(reg_base
(min_linear)(type_of(a'))
≤
a'
∧
a' + length(bl′)
≤
reg_size
(max_linear)(type_of(a')
THEN IF Mem?(type_of(a'))
THEN ap-
ply_side_effects
(split(min_page,
lin-
ear_read_side_effect_in_page)
(s)
ELSE mem-
ory_read_side_effect
(pm_phy`mem)
(a', bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr
(singleton(Read) ∪ singleton(Exe

Case splitting on null?(bl!1),

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1.1:`

```
{-1}  null?(bl′)
{-2}  Mem?(type_of(a′))
{-3}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{-4}  plain_memory?(pm_phy)
{-5}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-6}  is_linear_plain_memory?(pm′)
{-7}  pm′`states = pm_phy`states
{-8}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-9}  pm′`states(s′)
------------------------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                       IF null?(bl′) ∨
                                                          (reg_base
                                                             (min_linear)(type_of(a′))
                                                             ≤
                                                             a′
                                                             ∧
                                                             a′ + length(bl′)
                                                             ≤
                                                             reg_size
                                                             (max_linear)(type_of(a′)))
                                                         THEN IF Mem?(type_of(a′))
                                                                 THEN ap-
      ply_side_effects
                                                                        (split(min_page, a′, bl′),
                                                                           lin-
      ear_read_side_effect_in_page)
                                                                        (s)
                                                                 ELSE mem-
      ory_read_side_effect
                                                                        (pm_phy`mem)
                                                                        (a′, bl′, cp′)(s)
                                                                 ENDIF
                                                       ELSE Fatal
                                                       ENDIF)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                     (virt_to_phys_range(s′,
                                                         restrict
                                                            [Address, Mem-
      ory_Address_4G, boolean]
                                                            ((pm′`ro_addr ∪ pm′`rw_addr)),
                                                         (singleton(Read) ∪ singleton(Execute)))))
```

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of apply_side_effects,
Expanding the definition of ok_result,
Expanding the definition of reduce,

Using lemma split_null,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.1`.
`linear_plain_unchanged_memory_invariant.2.1.3.1.2`:

```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-8}  pm'`states(s')
------------------------------------------------------------------------------
{1}   null?(bl')
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                              Linear_memory
                                                              [Physical_memory, pm_p
                                                            IF null?(bl') ∨
                                                               (reg_base
                                                                 (min_linear)(type_of(a'))
                                                                 ≤
                                                                 a'
                                                                 ∧
                                                                 a' + length(bl')
                                                                 ≤
                                                                 reg_size
                                                                 (max_linear)(type_of(a'))
                                                             THEN IF Mem?(type_of(a'))
                                                                  THEN ap-
      ply_side_effects
                                                                       (split(min_page,
                                                                         lin-
      ear_read_side_effect_in_page)
                                                                       (s)
                                                                  ELSE mem-
      ory_read_side_effect
                                                                       (pm_phy`mem)
                                                                       (a', bl', cp')(s)
                                                                  ENDIF
                                                            ELSE Fatal
                                                            ENDIF)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                     (virt_to_phys_range(s',
                                                         restrict
                                                            [Address, Mem-
      ory_Address_4G, boolean]
                                                            ((pm'`ro_addr ∪ pm'`rw_addr
                                                         (singleton(Read) ∪ singleton(Exec
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1:`

{-1} Mem?(type_of(a'))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5} is_linear_plain_memory?(pm′)
{-6} pm′`states = pm_phy`states
{-7} (address_block(a', length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-8} reg_base(min_linear)(type_of(a')) ≤ a'
{-9} a' + length(bl′) ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{-10} pm′`states(s')

{1} null?(bl′)
{2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
apply_side_effects(split
(min_page,
a',
bl′),
linear_read_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))

Using lemma address_block_split_type,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1:`

{-1} is_linear_plain_memory?(pm′) ∧
(address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
⊃
every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))
{-2} Mem?(type_of(a′))
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-9} reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′‘states(s′)

{1} null?(bl′)
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_
apply_side_effects(split
(min_p
a′,
bl′),
lin-
ear_read_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_add
(singleton(Read) ∪ singleton(Ex

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma apply_side_effects_unchanged,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-3} every(λ (t: [Address, list[Byte]]):
Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
(split(min_page, a′, bl′))
{-4} Mem?(type_of(a′))
{-5} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
{-6} plain_memory?(pm_phy)
{-7} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-8} pm′`states = pm_phy`states
{-9} reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′`states(s′)

{1} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(apply_side_effects(split
(min_page, a′, bl′)
lin-
ear_read_side_effect_in_page))),
extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Mem-
ory_Address_4G, boolean]
((pm_phy`ro_addr ∪ pm_phy`rw_addr))))
{2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
apply_side_effects(split
(min_page,
a′,
bl′),
lin-
ear_read_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))
{3} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(linear_read_side_effect_in_page
(e))),
extend[Address, Mem-
ory_Address_4G, bool, FALSE]
(restrict[Address, Mem-
ory_Address_4G, boolean]
((pm_phy`ro_addr ∪ pm_phy`rw_addr)))))
(split(min_page, a′, bl′))
{4} null?(bl′)

Hiding formulas: (-7 1 2),

Using lemma split_linear_read_side_effects_unchanged_memory,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.1:`

```
{-1}  is_linear_plain_memory?(pm′) ∧
       (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
       ⊃
       every(λ (e: [Memory_Address_4G, list[Byte]]):
                 unchanged_memory_invariant?(pm_phy`mem, pm′`states,
                                             singleton(expr_2_super(linear_read_side_effect_in_pa
                                                                    (e))),
                                             (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
             (split(min_page, a′, bl′))
{-2}  is_linear_plain_memory?(pm′)
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-4}  every(λ (t: [Address, list[Byte]]):
                 Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
           (split(min_page, a′, bl′))
{-5}  Mem?(type_of(a′))
{-6}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-7}  plain_memory?(pm_phy)
{-8}  pm′`states = pm_phy`states
{-9}  reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′`states(s′)
-------------------------------------------------------------------------------
 {1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                 unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                             singleton(expr_2_super(linear_read_side_effect_in_pag
                                                                    (e))),
                                             extend[Address, Mem-
      ory_Address_4G, bool, FALSE]
                                                    (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                             ((pm_phy`ro_addr ∪ pm_phy`rw_addr))))
           (split(min_page, a′, bl′))
 {2}  null?(bl′)
```

Simplifying, rewriting, and recording with decision procedures,

Using lemma every_implied,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.2:`

```
  {-1}   is_linear_plain_memory?(pm′)
  {-2}   (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
  {-3}   every(λ (t: [Address, list[Byte]]):
                   Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
             (split(min_page, a′, bl′))
  {-4}   Mem?(type_of(a′))
  {-5}   (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
  {-6}   plain_memory?(pm_phy)
  {-7}   pm′`states = pm_phy`states
  {-8}   reg_base(min_linear)(type_of(a′)) ≤ a′
  {-9}   a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
 {-10}   pm′`states(s′)
--------------------------------------------------------------------------------
  {1}    0 ≤ a′`offset ∧ a′`offset < max_linear_offset
  {2}    every(λ (e: [Memory_Address_4G, list[Byte]]):
                   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                               singleton(expr_2_super(linear_read_side_effect_in_page
                                                                      (e))),
                                               extend[Address, Mem-
         ory_Address_4G, bool, FALSE]
                                                    (restrict[Address, Mem-
         ory_Address_4G, boolean]
                                                             ((pm_phy`ro_addr ∪ pm_phy`rw_addr)))))
             (split(min_page, a′, bl′))
  {3}    null?(bl′)
```

Keeping (-8 -9 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.2:`

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(apply_side_effects(split
(min_pag
lin-
ear_read_side_effect_in_page))),
extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Mem-
ory_Address_4G, boolean]
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))))
{-2} is_linear_plain_memory?(pm′)
{-3} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a', bl′))
{-5} Mem?(type_of(a'))
{-6} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-7} plain_memory?(pm_phy)
{-8} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-9} pm′‘states = pm_phy‘states
{-10} reg_base(min_linear)(type_of(a')) ≤ a'
{-11} a' + length(bl′) ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{-12} pm′‘states(s')

{1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_
apply_side_effects(split
(min_p
a',
bl′),
lin-
ear_read_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_add
(singleton(Read) ∪ singleton(Ex
{3} null?(bl′)

Keeping (-6 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (x!1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.3:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(apply_side_effects(split
                                                                              (min_page, a′, bl′)
                                                                              lin-
      ear_read_side_effect_in_page))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                ((pm_phy`ro_addr ∪ pm_phy`rw_addr))))
{-2}  is_linear_plain_memory?(pm′)
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-4}  every(λ (t: [Address, list[Byte]]):
              Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
        (split(min_page, a′, bl′))
{-5}  Mem?(type_of(a′))
{-6}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
{-7}  plain_memory?(pm_phy)
{-8}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-9}  pm′`states = pm_phy`states
{-10} reg_base(min_linear)(type_of(a′)) ≤ a′
{-11} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-12} pm′`states(s′)
-------------------------------------------------------------------------------
 {1}  (singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]) : apply_side_effects(split(min_p
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                             Linear_memory
                                                               [Physical_memory, pm_phy]):
                                                           apply_side_effects(split
                                                                                (min_page,
                                                                                  a′,
                                                                                  bl′),
                                                                                lin-
      ear_read_side_effect_in_page)
                                                                              (s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s′,
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm′`ro_addr ∪ pm′`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
 {3}  null?(bl′)
```

Keeping (1) and hiding *,

Expanding the definition of expr_2_super,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.1.3`.

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.2:`

```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-8}  reg_base(min_linear)(type_of(a')) ≤ a'
{-9}  a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{-10} pm'`states(s')
------------------------------------------------------------------------------------------------
{1}   0 ≤ a'`offset ∧ a'`offset < max_linear_offset
{2}   null?(bl')
{3}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                                Linear_memory
                                                                [Physical_memory, pm_
                                                          apply_side_effects(split
                                                                             (min_p
                                                                               a',
                                                                               bl'),
                                                                             lin-
      ear_read_side_effect_in_page)
                                                                            (s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_add
                                                          (singleton(Read) ∪ singleton(Ex
```

Keeping (-8 -9 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.2:`

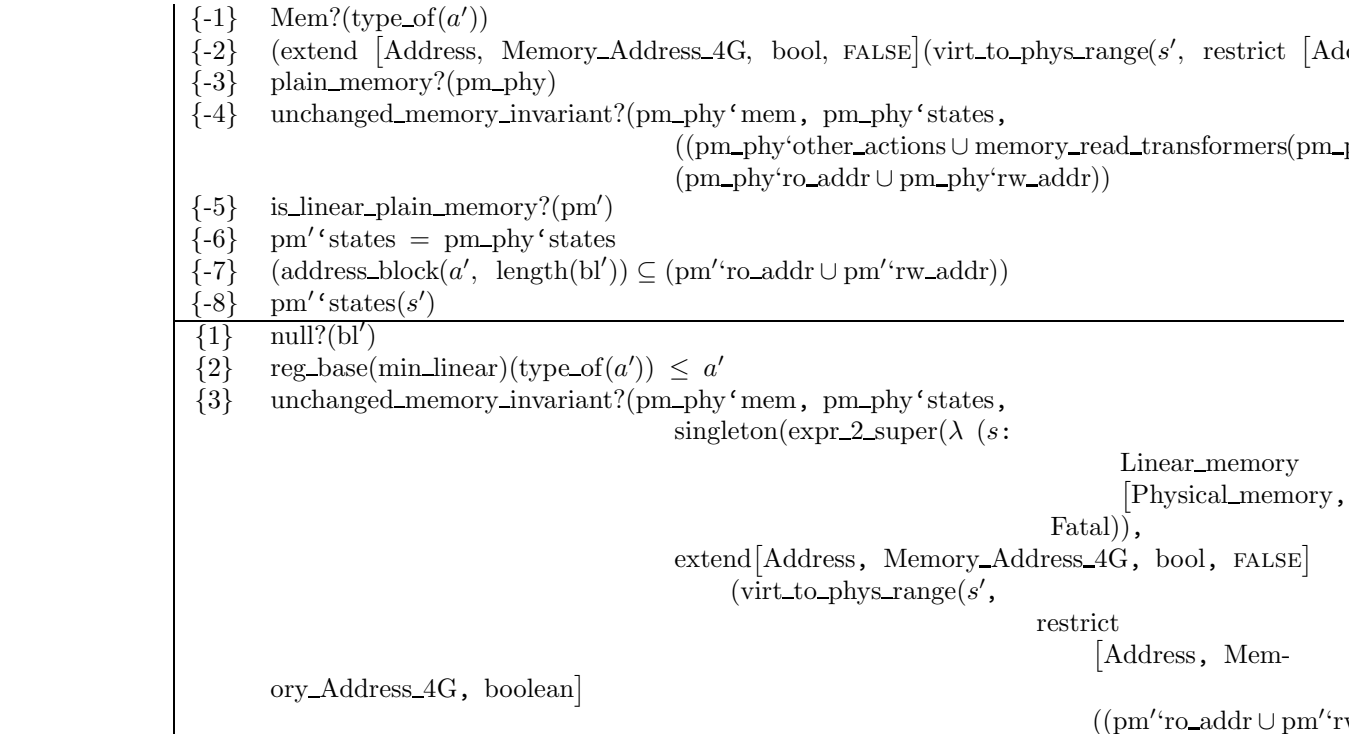
```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-8}  pm'`states(s')
------------------------------------------------------------------------------------------------------
{1}   null?(bl')
{2}   a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{3}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                             Linear_memory
                                                             [Physical_memory, pm_phy]):
                                                           Fatal)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Keeping (3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.2`.

`linear_plain_unchanged_memory_invariant.2.1.3.1.2.3:`



Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-7 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.1.2.3`.

`linear_plain_unchanged_memory_invariant.2.1.3.2:`

```
{-1}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-2}  plain_memory?(pm_phy)
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_pl
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'`states = pm_phy`states
{-6}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-7}  pm'`states(s')
------------------------------------------------------------------------------------------------------
 {1}  Mem?(type_of(a'))
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                       IF null?(bl') ∨
                                                          (reg_base
                                                             (min_linear)(type_of(a'))
                                                             ≤
                                                             a'
                                                             ∧
                                                             a' + length(bl')
                                                             ≤
                                                             reg_size
                                                             (max_linear)(type_of(a')))
                                                       THEN IF Mem?(type_of(a'))
                                                            THEN ap-
      ply_side_effects
                                                                 (split(min_page, a', bl'),
                                                                    lin-
      ear_read_side_effect_in_page)
                                                                 (s)
                                                            ELSE mem-
      ory_read_side_effect
                                                                 (pm_phy`mem)
                                                                 (a', bl', cp')(s)
                                                            ENDIF
                                                       ELSE Fatal
                                                       ENDIF)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                     (virt_to_phys_range(s',
                                                         restrict
                                                            [Address, Mem-
      ory_Address_4G, boolean]
                                                            ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                         (singleton(Read) ∪ singleton(Execute)))))
```

Using lemma unchanged_memory_invariant_mono,

Case splitting on subset?(singleton(expr_2_super(LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_read_side_effect(pm_phy`mem) (a!1, bl!1, cp!1)(s))), union(union(pm_phy`other_actions, memory_read_transformers(pm_phy`mem, union (pm_phy`ro_addr, pm_phy`rw_addr))), union(memory_read_side_effect_super_t union (pm_phy`ro_addr, pm_phy`rw_addr)), memory_write_side_effect_super_transformers(pm_phy`mem,

pm_phy‘rw_addr)))),

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.3.2.1:

{-1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_effect(pm_p
{-2} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_effect(pm_p
∧
(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
∧
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
mem-
ory_read_side_effect(pm_phy‘mem)
(a′, bl′, cp′)(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_pl
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-9} pm′‘states(s′)

{1} Mem?(type_of(a′))
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
IF null?(bl′) ∨
(reg_base
(min_linear)(type_of(a′))
≤
a′
∧
a′ + length(bl′)
≤
reg_size
(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN ap-
ply_side_effects
(split(min_page, a′, bl′),
lin-
ear_read_side_effect_in_page)
(s)
ELSE mem-
ory_read_side_effect
(pm_phy‘mem)
(a′, bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.3.2.1.1:`

```
{-1}  (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_p
                                                         mem-
      ory_read_side_effect(pm_phy`mem)
                                                           (a', bl', cp')(s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr
                                                          (singleton(Read) ∪ singleton(Exec
{-5}  plain_memory?(pm_phy)
{-6}  is_linear_plain_memory?(pm')
{-7}  pm'`states = pm_phy`states
{-8}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-9}  pm'`states(s')
------------------------------------------------------------------------------
{1}   Mem?(type_of(a'))
{2}   null?(bl')
{3}   a' + length(bl') ≤
       reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{4}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_p
                                                         Fatal)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr
                                                          (singleton(Read) ∪ singleton(Exec
```

Keeping (4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.3.2.1.2:`

```
{-1}  (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_effect(pm_p
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_pl
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                         mem-
      ory_read_side_effect(pm_phy`mem)
                                                           (a', bl', cp')(s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
{-5}  plain_memory?(pm_phy)
{-6}  is_linear_plain_memory?(pm')
{-7}  pm'`states = pm_phy`states
{-8}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-9}  pm'`states(s')
------------------------------------------------------------------------------------------------------
{1}   Mem?(type_of(a'))
{2}   null?(bl')
{3}   reg_base(min_linear)(type_of(a')) ≤ a'
{4}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                         Fatal)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-8 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.3.2.2:`

```
{-1}  (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_
       ∧
       (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
        ∧
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`
                                    (pm_phy`ro_addr ∪ pm_phy`rw_addr))
       ⊃
       unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                   singleton(expr_2_super(λ (s:
                                                              Linear_memory
                                                              [Physical_memory, pm_ph
                                                            mem-
ory_read_side_effect(pm_phy`mem)
                                                              (a', bl', cp')(s))),
                                   extend[Address, Memory_Address_4G, bool, FALSE]
                                       (virt_to_phys_range(s',
                                                           restrict
                                                             [Address, Mem-
ory_Address_4G, boolean]
                                                             ((pm'`ro_addr ∪ pm'`rw_add
                                                           (singleton(Read) ∪ singleton(Ex
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-8}  pm'`states(s')
-------------------------------------------------------------------------------
{1}   (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_
{2}   Mem?(type_of(a'))
{3}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                             Linear_memory
                                                             [Physical_memory, pm_p
                                                           IF null?(bl') ∨
                                                              (reg_base
                                                                (min_linear)(type_of(a'))
                                                                ≤
                                                                a'
                                                                ∧
                                                                a' + length(bl')
                                                                ≤
                                                                reg_size
                                                                (max_linear)(type_of(a'))
                                                            THEN IF Mem?(type_of(a'))
                                                                 THEN ap-
ply_side_effects
                                                                        (split(min_page,
                                                                          lin-
ear_read_side_effect_in_page)
                                                                        (s)
                                                                 ELSE mem-
ory_read_side_effect
                                                                        (pm_phy`mem)
                                                                        (a', bl', cp')(s)
                                                                 ENDIF
                                                           ELSE Fatal
                                                           ENDIF)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
```

Hiding formulas: (-1 -4 3),
Case splitting on memory_read_side_effect_super_transformers(pm_phy‘mem, union (pm_phy‘ro_addr, pm_phy‘rw_addr)) (expr_2_super (LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_read_side_effect (pm_phy‘mem) (a!1, bl!1, cp!1)(s))),
we get 2 subgoals:
`linear_plain_unchanged_memory_invariant.2.1.3.2.2.1:`

{-1} memory_read_side_effect_super_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
mem-
ory_read_side_effect(pm_phy‘mem)
(a′, bl′, cp′)(s)))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{-3} plain_memory?(pm_phy)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states = pm_phy‘states
{-6} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7} pm′‘states(s′)

{1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_effect(pm_p
{2} Mem?(type_of(a′))

Keeping (-1 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.2.1`.
`linear_plain_unchanged_memory_invariant.2.1.3.2.2.2:`

{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{-2} plain_memory?(pm_phy)
{-3} is_linear_plain_memory?(pm′)
{-4} pm′‘states = pm_phy‘states
{-5} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6} pm′‘states(s′)

{1} memory_read_side_effect_super_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
mem-
ory_read_side_effect(pm_phy‘mem)
(a′, bl′, cp′)(s)))
{2} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_read_side_effect(pm_p
{3} Mem?(type_of(a′))

Hiding formulas: 2,
Expanding the definition of memory_read_side_effect_super_transformers,
Instantiating quantified variables,
Expanding the definition of expr_2_super,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of member,
Expanding the definition of address_block,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.3.2.2.2`.

`linear_plain_unchanged_memory_invariant.2.1.4:`

```
{-1}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-2}  plain_memory?(pm_phy)
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'`states = pm_phy`states
{-6}  memory_write_side_effect_super_transformers(pm'`mem, pm'`rw_addr)(q')
{-7}  pm'`states(s')
------------------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states, singleton(q'),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr
                                                          (singleton(Read) ∪ singleton(Exec
```

Expanding the definition of memory_write_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -8,

Hiding formulas: (-1 -8),

Rewriting using pm_write_side_effect_linear, matching in *,

Expanding the definition of linear_write_side_effect,

Installing automatic rewrites from: union_right

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1:`

```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-8}  pm'`states(s')
------------------------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                       IF null?(bl') ∨
                                                          (reg_base
                                                             (min_linear)(type_of(a'))
                                                             ≤
                                                             a'
                                                             ∧
                                                             a' + length(bl')
                                                             ≤
                                                             reg_size
                                                             (max_linear)(type_of(a')))
                                                       THEN IF Mem?(type_of(a'))
                                                            THEN ap-
      ply_side_effects
                                                                 (split(min_page, a', bl'),
                                                                    lin-
      ear_write_side_effect_in_page)
                                                                 (s)
                                                            ELSE mem-
      ory_write_side_effect
                                                                 (pm_phy`mem)
                                                                 (a', bl', cp')(s)
                                                            ENDIF
                                                       ELSE Fatal
                                                       ENDIF)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                     (virt_to_phys_range(s',
                                                         restrict
                                                            [Address, Mem-
      ory_Address_4G, boolean]
                                                            ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                         (singleton(Read) ∪ singleton(Execute)))))
```

Case splitting on null?(bl!1),

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.1:`

{-1} null?(bl′)
{-2} Mem?(type_of(a′))
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘me
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9} pm′‘states(s′)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
IF null?(bl′) ∨
(reg_base
(min_linear)(type_of(a′))
≤
a′
∧
a′ + length(bl′)
≤
reg_size
(max_linear)(type_of(a′)
THEN IF Mem?(type_of(a′))
THEN ap-
ply_side_effects
(split(min_page,
lin-
ear_write_side_effect_in_page)
(s)
ELSE mem-
ory_write_side_effect
(pm_phy‘mem)
(a′, bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr
(singleton(Read) ∪ singleton(Exec

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of apply_side_effects,
Expanding the definition of ok_result,
Expanding the definition of reduce,

Using lemma split_null,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.1`.
`linear_plain_unchanged_memory_invariant.2.1.4.1.2`:

```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-8}  pm'`states(s')
------------------------------------------------------------------------------
{1}   null?(bl')
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                       IF null?(bl') ∨
                                                          (reg_base
                                                             (min_linear)(type_of(a'))
                                                             ≤
                                                             a'
                                                             ∧
                                                             a' + length(bl')
                                                             ≤
                                                             reg_size
                                                             (max_linear)(type_of(a')))
                                                        THEN IF Mem?(type_of(a'))
                                                             THEN ap-
      ply_side_effects
                                                                  (split(min_page, a', bl'),
                                                                    lin-
      ear_write_side_effect_in_page)
                                                                  (s)
                                                             ELSE mem-
      ory_write_side_effect
                                                                  (pm_phy`mem)
                                                                  (a', bl', cp')(s)
                                                             ENDIF
                                                       ELSE Fatal
                                                       ENDIF)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                     (virt_to_phys_range(s',
                                                         restrict
                                                           [Address, Mem-
      ory_Address_4G, boolean]
                                                           ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                         (singleton(Read) ∪ singleton(Execute)))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1:`

```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`m
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-8}  reg_base(min_linear)(type_of(a')) ≤ a'
{-9}  a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{-10} pm'`states(s')
------------------------------------------------------------------------------
 {1}  null?(bl')
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                              Linear_memory
                                                              [Physical_memory, pm_
                                                            apply_side_effects(split
                                                                                 (min_p
                                                                                   a',
                                                                                   bl'),
                                                                                 lin-
      ear_write_side_effect_in_page)
                                                                               (s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                                   ((pm'`ro_addr ∪ pm'`rw_add
                                                                  (singleton(Read) ∪ singleton(Ex
```

Using lemma address_block_split_type,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1:`

{-1} is_linear_plain_memory?(pm′) ∧
(address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
⊃
every(λ (t: [Address, list[Byte]]):
Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
(split(min_page, a′, bl′))
{-2} Mem?(type_of(a′))
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-6} is_linear_plain_memory?(pm′)
{-7} pm′`states = pm_phy`states
{-8} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-9} reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′`states(s′)

{1} null?(bl′)
{2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
apply_side_effects(split
(min_page,
a′,
bl′),
linear_write_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1:

{-1} is_linear_plain_memory?(pm′)
{-2} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))
{-3} Mem?(type_of(a′))
{-4} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-5} plain_memory?(pm_phy)
{-6} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9} reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′‘states(s′)

{1} null?(bl′)
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_
apply_side_effects(split
(min_p
a′,
bl′),
lin-
ear_write_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_add
(singleton(Read) ∪ singleton(Ex

Using lemma apply_side_effects_unchanged,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))
{-3} Mem?(type_of(a′))
{-4} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
{-5} plain_memory?(pm_phy)
{-6} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9} reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′‘states(s′)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(apply_side_effects(split
(min_page, a′, bl′)
lin-
ear_write_side_effect_in_page))),
extend[Address, Memory_Address_4G, bool, FALSE]
(restrict[Address, Mem-
ory_Address_4G, boolean]
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))))
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
apply_side_effects(split
(min_page,
a′,
bl′),
lin-
ear_write_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))
{3} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(linear_write_side_effect_in_page
(e))),
extend[Address, Mem-
ory_Address_4G, bool, FALSE]
(restrict[Address, Mem-
ory_Address_4G, boolean]
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))))
(split(min_page, a′, bl′))
{4} null?(bl′)

Hiding formulas: (-7 1 2),

Using lemma split_linear_write_side_effects_unchanged_memory,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.1:`

```
{-1}  is_linear_plain_memory?(pm′) ∧ (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr) ⊃
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
                                            singleton(expr_2_super(linear_write_side_effect_in_p
                                                                   (e))),
                                            (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
              (split(min_page, a′, bl′))
{-2}  is_linear_plain_memory?(pm′)
{-3}  every(λ (t: [Address, list[Byte]]):
              Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
            (split(min_page, a′, bl′))
{-4}  Mem?(type_of(a′))
{-5}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-6}  plain_memory?(pm_phy)
{-7}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-8}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9}  reg_base(min_linear)(type_of(a′)) ≤ a′
{-10} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-11} pm′‘states(s′)
-------------------------------------------------------------------------------
 {1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                          singleton(expr_2_super(linear_write_side_effect_in_pa
                                                                 (e))),
                                          extend[Address, Mem-
      ory_Address_4G, bool, FALSE]
                                                 (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                           ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))))
            (split(min_page, a′, bl′))
 {2}  null?(bl′)
```

Simplifying, rewriting, and recording with decision procedures,

Using lemma every_implied,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.2:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  every(λ (t: [Address, list[Byte]]):
                  Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
              (split(min_page, a′, bl′))
  {-3}  Mem?(type_of(a′))
  {-4}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
  {-5}  plain_memory?(pm_phy)
  {-6}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                    ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
                                    (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
  {-7}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
  {-8}  reg_base(min_linear)(type_of(a′)) ≤ a′
  {-9}  a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
 {-10}  pm′‘states(s′)
  ------------------------------------------------------------------------------------------------
   {1}  0 ≤ a′‘offset ∧ a′‘offset < max_linear_offset
   {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                              singleton(expr_2_super(linear_write_side_effect_in_page
                                                                     (e))),
                                              extend[Address, Mem-
        ory_Address_4G, bool, FALSE]
                                                     (restrict[Address, Mem-
        ory_Address_4G, boolean]
                                                              ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))))
              (split(min_page, a′, bl′))
   {3}  null?(bl′)
```

Keeping (-8 -9 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.2:`

```
{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(apply_side_effects(split
                                                                            (min_pag
                                                                            lin-
      ear_write_side_effect_in_page))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))))
{-2}  is_linear_plain_memory?(pm′)
{-3}  every(λ (t: [Address, list[Byte]]):
              Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
        (split(min_page, a′, bl′))
{-4}  Mem?(type_of(a′))
{-5}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-6}  plain_memory?(pm_phy)
{-7}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-8}  pm′‘states = pm_phy‘states
{-9}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-10} reg_base(min_linear)(type_of(a′)) ≤ a′
{-11} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-12} pm′‘states(s′)
-------------------------------------------------------------------------------
 {1}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
 {2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(λ (s:
                                                                Linear_memory
                                                                [Physical_memory, pm_
                                                           apply_side_effects(split
                                                                              (min_p
                                                                                a′,
                                                                                bl′),
                                                                              lin-
      ear_write_side_effect_in_page)
                                                                             (s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s′,
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm′‘ro_addr ∪ pm′‘rw_add
                                                          (singleton(Read) ∪ singleton(Ex
 {3}  null?(bl′)
```

Keeping (-5 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (x!1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.3:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(apply_side_effects(split
                                                                            (min_page, a', bl')
                                                                            lin-
      ear_write_side_effect_in_page))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                ((pm_phy`ro_addr ∪ pm_phy`rw_addr))))
{-2}  is_linear_plain_memory?(pm')
{-3}  every(λ (t: [Address, list[Byte]]):
              Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
          (split(min_page, a', bl'))
{-4}  Mem?(type_of(a'))
{-5}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
{-6}  plain_memory?(pm_phy)
{-7}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-8}  pm'`states = pm_phy`states
{-9}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-10} reg_base(min_linear)(type_of(a')) ≤ a'
{-11} a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{-12} pm'`states(s')
------------------------------------------------------------------------------
 {1}  (singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]) : apply_side_effects(split(min_p
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                             [Physical_memory, pm_phy]):
                                                         apply_side_effects(split
                                                                            (min_page,
                                                                              a',
                                                                              bl'),
                                                                            lin-
      ear_write_side_effect_in_page)
                                                                           (s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                            [Address, Mem-
      ory_Address_4G, boolean]
                                                            ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
 {3}  null?(bl')
```

Keeping (1) and hiding *,

Expanding the definition of expr_2_super,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.1.3`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} Mem?(type_of(a′))
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘m
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6} pm′‘states = pm_phy‘states
{-7} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-8} reg_base(min_linear)(type_of(a′)) ≤ a′
{-9} a′ + length(bl′) ≤ (#type_of := type_of(a′), offset := max_linear_offset#)
{-10} pm′‘states(s′)

{1} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{2} null?(bl′)
{3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_
apply_side_effects(split
(min_p
a′,
bl′),
lin-
ear_write_side_effect_in_page)
(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_add
(singleton(Read) ∪ singleton(Ex

Keeping (-7 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1‘rw_addr, c gets union(pm!1‘ro_addr, pm!1‘rw_addr),

Rewriting using union_subset3, matching in *,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.2:`

```
  {-1}   Mem?(type_of(a'))
  {-2}   (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_
  {-3}   plain_memory?(pm_phy)
  {-4}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                     ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                     (pm_phy`ro_addr ∪ pm_phy`rw_addr))
  {-5}   is_linear_plain_memory?(pm')
  {-6}   pm'`states = pm_phy`states
  {-7}   (address_block(a', length(bl')) ⊆ pm'`rw_addr)
  {-8}   reg_base(min_linear)(type_of(a')) ≤ a'
  {-9}   a' + length(bl') ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
  {-10}  pm'`states(s')
  ------------------------------------------------------------------------------------------------------
  {1}    0 ≤ a'`offset ∧ a'`offset < max_linear_offset
  {2}    null?(bl')
  {3}    unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                     singleton(expr_2_super(λ (s:
                                                                Linear_memory
                                                                [Physical_memory, pm_phy]):
                                                            apply_side_effects(split
                                                                               (min_page,
                                                                                a',
                                                                                bl'),
                                                                               lin-
         ear_write_side_effect_in_page)
                                                                              (s))),
                                     extend[Address, Memory_Address_4G, bool, FALSE]
                                         (virt_to_phys_range(s',
                                                             restrict
                                                                 [Address, Mem-
         ory_Address_4G, boolean]
                                                                 ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                             (singleton(Read) ∪ singleton(Execute)))))
```

Keeping (-8 -9 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.1.2`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.2:`

{-1} Mem?(type_of(a'))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5} is_linear_plain_memory?(pm$'$)
{-6} pm$'$`states = pm_phy`states
{-7} (address_block(a', length(bl$'$)) ⊆ pm$'$`rw_addr)
{-8} pm$'$`states(s')

{1} null?(bl$'$)
{2} a' + length(bl$'$) ≤ (#type_of := type_of(a'), offset := max_linear_offset#)
{3} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
Fatal)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm$'$`ro_addr ∪ pm$'$`rw_addr
(singleton(Read) ∪ singleton(Exe

Keeping (3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.2`.

`linear_plain_unchanged_memory_invariant.2.1.4.1.2.3:`

```
{-1}  Mem?(type_of(a'))
{-2}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address, Memory_A
{-3}  plain_memory?(pm_phy)
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_ph
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-8}  pm'`states(s')
------------------------------------------------------------------------------
{1}   null?(bl')
{2}   reg_base(min_linear)(type_of(a')) ≤ a'
{3}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                             Linear_memory
                                                             [Physical_memory, pm_phy]):
                                                           Fatal)),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm'`ro_addr ∪ pm'`rw_addr)),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-7 1 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.1.2.3`.

`linear_plain_unchanged_memory_invariant.2.1.4.2:`

{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s', restrict [Address,
{-2} plain_memory?(pm_phy)
{-3} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`me
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-4} is_linear_plain_memory?(pm′)
{-5} pm′`states = pm_phy`states
{-6} (address_block(a', length(bl′)) ⊆ pm′`rw_addr)
{-7} pm′`states(s')

{1} Mem?(type_of(a'))
{2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
IF null?(bl′) ∨
(reg_base
(min_linear)(type_of(a'))
≤
a'
∧
a' + length(bl′)
≤
reg_size
(max_linear)(type_of(a')
THEN IF Mem?(type_of(a'))
THEN ap-
ply_side_effects
(split(min_page,
lin-
ear_write_side_effect_in_page)
(s)
ELSE mem-
ory_write_side_effect
(pm_phy`mem)
(a', bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr
(singleton(Read) ∪ singleton(Exec

Using lemma unchanged_memory_invariant_mono,

Case splitting on subset?(singleton(expr_2_super(LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_write_side_effect(pm_phy`mem) (a!1, bl!1, cp!1)(s))), union(union(pm_phy`other_actions, memory_read_transformers(pm_phy`mem, union (pm_phy`ro_addr, pm_phy`rw_addr))), union(memory_read_side_ef union (pm_phy`ro_addr, pm_phy`rw_addr)), memory_write_side_effect_super_transformers(pm_phy`mem,

pm_phy‘rw_addr)))),

we get 2 subgoals:

linear_plain_unchanged_memory_invariant.2.1.4.2.1:

{-1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_
{-2} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_
∧
(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
∧
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_ph
mem-
ory_write_side_effect(pm_phy‘mem)
(a′, bl′, cp′)(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_add
(singleton(Read) ∪ singleton(Ex
{-3} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-4} plain_memory?(pm_phy)
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘me
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9} pm′‘states(s′)

{1} Mem?(type_of(a′))
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
IF null?(bl′) ∨
(reg_base
(min_linear)(type_of(a′))
≤
a′
∧
a′ + length(bl′)
≤
reg_size
(max_linear)(type_of(a′)
THEN IF Mem?(type_of(a′))
THEN ap-
ply_side_effects
(split(min_page,
lin-
ear_write_side_effect_in_page)
(s)
ELSE mem-
ory_write_side_effect
(pm_phy‘mem)
(a′, bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant.2.1.4.2.1.1:`

{-1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_effect(pm_p
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{-3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_pl
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
mem-
ory_write_side_effect(pm_phy‘mem)
(a′, bl′, cp′)(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))
{-5} plain_memory?(pm_phy)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9} pm′‘states(s′)

{1} Mem?(type_of(a′))
{2} null?(bl′)
{3} a′ + length(bl′) ≤
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
Fatal)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))

Keeping (4) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.1.1`.

`linear_plain_unchanged_memory_invariant.2.1.4.2.1.2:`

{-1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘me
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
mem-
ory_write_side_effect(pm_phy‘mem
(a′, bl′, cp′)(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr
(singleton(Read) ∪ singleton(Exec
{-5} plain_memory?(pm_phy)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9} pm′‘states(s′)

{1} Mem?(type_of(a′))
{2} null?(bl′)
{3} reg_base(min_linear)(type_of(a′)) ≤ a′
{4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
Fatal)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr
(singleton(Read) ∪ singleton(Exec

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-8 1 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.1.2`.

linear_plain_unchanged_memory_invariant.2.1.4.2.2:

{-1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_effect(pm_p
∧
(extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
∧
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
mem-
ory_write_side_effect(pm_phy‘mem)
(a′, bl′, cp′)(s))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
(singleton(Read) ∪ singleton(Execute)))))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_pl
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-5} is_linear_plain_memory?(pm′)
{-6} pm′‘states = pm_phy‘states
{-7} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-8} pm′‘states(s′)

{1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_effect(pm_p
{2} Mem?(type_of(a′))
{3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
IF null?(bl′) ∨
(reg_base
(min_linear)(type_of(a′))
≤
a′
∧
a′ + length(bl′)
≤
reg_size
(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN ap-
ply_side_effects
(split(min_page, a′, bl′),
lin-
ear_write_side_effect_in_page)
(s)
ELSE mem-
ory_write_side_effect
(pm_phy‘mem)
(a′, bl′, cp′)(s)
ENDIF
ELSE Fatal
ENDIF)),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,

Hiding formulas: (-1 -4 3),
Case splitting on memory_write_side_effect_super_transformers(pm_phy‘mem, pm_phy‘rw_addr)(expr_2_super (LAMBDA (s: Linear_memory [Physical_memory, pm_phy]): memory_write_side_effect(pm_phy‘mem) (a!1, bl!1, cp!1)(s))),
we get 2 subgoals:
`linear_plain_unchanged_memory_invariant.2.1.4.2.2.1:`

{-1} memory_write_side_effect_super_transformers(pm_phy‘mem, pm_phy‘rw_addr)(expr_2_super(λ (s: Linear_memory [Physical_memory, pmem-ory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s)))
{-2} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-3} plain_memory?(pm_phy)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states = pm_phy‘states
{-6} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-7} pm′‘states(s′)

{1} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_
{2} Mem?(type_of(a′))

Keeping (-1 -6 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.2.1`.
`linear_plain_unchanged_memory_invariant.2.1.4.2.2.2:`

{-1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{-2} plain_memory?(pm_phy)
{-3} is_linear_plain_memory?(pm′)
{-4} pm′‘states = pm_phy‘states
{-5} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-6} pm′‘states(s′)

{1} memory_write_side_effect_super_transformers(pm_phy‘mem, pm_phy‘rw_addr)(expr_2_super(λ (s: Linear_memory [Physical_memory, pmem-ory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s)))
{2} (singleton(expr_2_super(λ (s : Linear_memory[Physical_memory, pm_phy]) : memory_write_side_
{3} Mem?(type_of(a′))

Hiding formulas: 2,
Expanding the definition of memory_write_side_effect_super_transformers,
Instantiating quantified variables,
Expanding the definition of expr_2_super,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of member,
Expanding the definition of address_block,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.1.4.2.2.2`.
`linear_plain_unchanged_memory_invariant.2.2`:

```
{-1}  plain_memory?(pm_phy) ⊃
        unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                    ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
                                    (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}  is_linear_plain_memory?(pm′) ⊃ pm′‘states = pm_phy‘states
{-3}  is_linear_plain_memory?(pm′) ⊃ plain_memory?(pm_phy)
{-4}  (pm′‘other_actions(q′) ∨
         memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr))(q′))
       ∨
       memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr))(q′)
        ∨ memory_write_side_effect_super_transformers(pm′‘mem, pm′‘rw_addr)(q′)
{-5}  pm′‘states(s′)
{-6}  is_linear_plain_memory?(pm′)
------------------------------------------------------------------------------------------
{1}   (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(q′),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s′,
                                                        restrict
                                                          [Address, Mem-
ory_Address_4G, boolean]
                                                          ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                                                        (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: (-1 -2 -3 -4 2),
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1) and hiding *,
Case splitting on union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)),
we get 2 subgoals:
`linear_plain_unchanged_memory_invariant.2.2.1`:

```
{-1}  (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Address_4G, b
        = restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{-2}  (virt_to_phys_range(s′, (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address,
------------------------------------------------------------------------------------------
{1}   (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
```

Replacing using formula -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant.2.2.1`.
`linear_plain_unchanged_memory_invariant.2.2.2`:

```
{-1}  (virt_to_phys_range(s′, (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address,
------------------------------------------------------------------------------------------
{1}   (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Address_4G, b
        = restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{2}   (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_A
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant.2.2.2`.

Q.E.D.

C.116.3 Linear_Memory_Blessing_Properties.linear_plain_unchanged_memory_invariant_w

Terse proof for `linear_plain_unchanged_memory_invariant_write`.

`linear_plain_unchanged_memory_invariant_write:`

```
{1}  ∀ (pm: Plain_Memory[Physical_memory]):
       is_linear_plain_memory?(pm) ⊃
         unchanged_memory_invariant?(pm‘mem, pm‘states, mem-
     ory_write_transformers(pm‘mem, pm‘rw_addr),
                                                pm‘ro_addr)
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: has_next_state singleton subset_reflexive pm_write_linear linear_write Mem in_memory Mem reg_size union_left max_linear < min_linear Address_Helpers.<= reg_base

Using lemma linear_unchanged_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant_write.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                    memory_write_transformers(pm′‘mem, pm′‘rw_addr),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                        (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                  (pm′‘ro_addr)))
-------------------------------------------------------------------------------
{1}  unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                    memory_write_transformers(pm′‘mem, pm′‘rw_addr), pm′‘ro
```

Case splitting on extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr)) = pm!1‘ro_addr,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant_write.1.1:`

```
{-1}  extend[Address, Memory_Address_4G, bool, FALSE]
          (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr))
       = pm'`ro_addr
{-2}  is_linear_plain_memory?(pm')
{-3}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem, pm'`rw_addr),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                (pm'`ro_addr)))
-------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem, pm'`rw_addr), pm'`ro_addr)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.1.1`.
`linear_plain_unchanged_memory_invariant_write.1.2:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem, pm'`rw_addr),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Mem-
      ory_Address_4G, boolean]
                                                (pm'`ro_addr)))
-------------------------------------------------------------------------------
{1}   extend[Address, Memory_Address_4G, bool, FALSE]
          (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr))
       = pm'`ro_addr
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem, pm'`rw_addr), pm'`ro_addr)
```

Hiding formulas: (-2 2),
Expanding the definition of extend,
Expanding the definition of restrict,
Applying decompose-equality,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.1.2`.
`linear_plain_unchanged_memory_invariant_write.2:`

```
{-1}  is_linear_plain_memory?(pm')
-------------------------------------------------------------------------------
{1}   ∀ (s: (pm'`states)):
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    memory_write_transformers(pm'`mem, pm'`rw_addr),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s,
                                                            restrict
                                                                [Address, Mem-
      ory_Address_4G, boolean]
                                                                (pm'`ro_addr),
                                                            (singleton(Read) ∪ singleton(Execute)))))
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem, pm'`rw_addr), pm'`ro_addr)
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using unchanged_memory_invariant_all_transformers, matching in * where pm gets pm_phy‘mem,
Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of linear_write,
Hiding formulas: (-3),
Using lemma pm_memory_addr,
Installing automatic rewrites from: union_right
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_plain_phy,
Rewriting using plain_memory_unchanged_composition[Physical_memory, Address, Unit], matching in * where pm gets pm_phy, addresses gets extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s!1, restrict[Address, Memory_Address_4G, boolean] (pm!1‘ro_addr), union(singleton(Read), singleton(Execute)))), P gets LAMBDA (a: Address): FORALL (s: (pm_phy‘states)): OK?(linear_resolve(a!1, Write)(s)) IMPLIES data(linear_resolve(a!1, Write)(s)) = a,
we get 4 subgoals:
`linear_plain_unchanged_memory_invariant_write.2.1:`

{-1} plain_memory?(pm_phy)
{-2} is_linear_plain_memory?(pm′)
{-3} Mem?(type_of(a′))
{-4} 0 ≤ a′‘offset
{-5} a′‘type_of = a′‘type_of
{-6} a′‘offset < max_linear_offset
{-7} b′ < max_byte
{-8} pm′‘rw_addr(a′)
{-9} pm′‘states(s′)

{1} (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address,
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(linear_resolve(a′, Write) ##
(λ (pa: Address):
mem-
ory_write(pm_phy‘mem)
(pa, b′)))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Mem-
ory_Address_4G, boolean]
(pm′‘ro_addr),
(singleton(Read) ∪ singleton(Exe

Hiding formulas: 2,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-3 1) and hiding *,
Installing automatic rewrites from: subset? member extend restrict
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -1 with the terms: (x!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.1`.
`linear_plain_unchanged_memory_invariant_write.2.2`:

```
{-1}  plain_memory?(pm_phy)
{-2}  is_linear_plain_memory?(pm')
{-3}  Mem?(type_of(a'))
{-4}  0 ≤ a'`offset
{-5}  a'`type_of = a'`type_of
{-6}  a'`offset < max_linear_offset
{-7}  b' < max_byte
{-8}  pm'`rw_addr(a')
{-9}  pm'`states(s')
------------------------------------------------------------------------
{1}   ∀ (s_1: (pm_phy`states)):
        OK?(linear_resolve(a', Write)(s_1)) ⊃
         (∀ (s: (pm_phy`states)):
            OK?(linear_resolve(a', Write)(s)) ⊃
             data(linear_resolve(a', Write)(s)) = data(linear_resolve(a', Write)(s_1)))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(a', Write) ##
                                                         (λ (pa: Address):
                                                            mem-
ory_write(pm_phy`mem)
                                                                (pa, b')))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
ory_Address_4G, boolean]
                                                              (pm'`ro_addr),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma linear_resolve_same_result,
Using lemma pm_states,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.2`.

`linear_plain_unchanged_memory_invariant_write.2.3:`

```
{-1}  plain_memory?(pm_phy)
{-2}  is_linear_plain_memory?(pm')
{-3}  Mem?(type_of(a'))
{-4}  0 ≤ a'`offset
{-5}  a'`type_of = a'`type_of
{-6}  a'`offset < max_linear_offset
{-7}  b' < max_byte
{-8}  pm'`rw_addr(a')
{-9}  pm'`states(s')
------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(a', Write))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s',
                                                        restrict
                                                          [Address, Memory_Address_4G, boolean]
                                                          (pm'`ro_addr),
                                                        (singleton(Read) ∪ singleton(Exec
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(a', Write) ##
                                                         (λ (pa: Address):
                                                            memory_write(pm_phy`mem)
                                                              (pa, b')))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s',
                                                        restrict
                                                          [Address, Memory_Address_4G, boolean]
                                                          (pm'`ro_addr),
                                                        (singleton(Read) ∪ singleton(Exec
```

Hiding formulas: 2,
Using lemma linear_resolve_unchanged_pm_phy,
Using lemma pm_states,
Using lemma unchanged_memory_invariant_mono[Physical_memory],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 2),
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Expanding the definition of member,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_resolve_address,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-6 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.3.1:`

{-1}	Mem?(lin_a′‘type_of)
{-2}	$0 \leq$ lin_a′‘offset
{-3}	lin_a′‘offset $<$ max_linear_offset
{-4}	restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr)(lin_a′)
{-5}	union(singleton(Read), singleton(Execute))(ac′)
{-6}	OK?(linear_resolve(lin_a′, ac′)(s′))
{-7}	data(linear_resolve(lin_a′, ac′)(s′)) $=$ x′
{1}	OK?(linear_resolve(lin_a′, ac′)(s′)) ∧ data(linear_resolve(lin_a′, ac′)(s′)) $=$ x′

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.3.1`.

`linear_plain_unchanged_memory_invariant_write.2.3.2:`

{-1}	Mem?(lin_a′‘type_of)
{-2}	$0 \leq$ lin_a′‘offset
{-3}	lin_a′‘offset $<$ max_linear_offset
{-4}	restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr)(lin_a′)
{-5}	union(singleton(Read), singleton(Execute))(ac′)
{-6}	OK?(linear_resolve(lin_a′, ac′)(s′))
{-7}	data(linear_resolve(lin_a′, ac′)(s′)) $=$ x′
{1}	restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))(lin_a′)

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.3.2`.

`linear_plain_unchanged_memory_invariant_write.2.4:`

{-1} plain_memory?(pm_phy)
{-2} is_linear_plain_memory?(pm′)
{-3} Mem?(type_of(a′))
{-4} 0 ≤ a′‘offset
{-5} a′‘type_of = a′‘type_of
{-6} a′‘offset < max_linear_offset
{-7} b′ < max_byte
{-8} pm′‘rw_addr(a′)
{-9} pm′‘states(s′)

{1} ∀ (d:
(λ (a: Address):
∀ (s: (pm_phy‘states)):
OK?(linear_resolve(a′, Write)(s)) ⊃
data(linear_resolve(a′, Write)(s)) = a)):
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)(d, b′)))
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Memory_Address_4G, boolean]
(pm′‘ro_addr),
(singleton(Read) ∪ singleton(E

{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(linear_resolve(a′, Write) ##
(λ (pa: Address):
memory_write(pm_phy‘mem)
(pa, b′)))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s′,
restrict
[Address, Memory_Address_4G, boolean]
(pm′‘ro_addr),
(singleton(Read) ∪ singleton(Exe

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma pm_states,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Installing automatic rewrites from: pm_linear_resolve_write_ok
Using lemma pm_plain_phy,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_unchanged_memory_invariant_write,
Using lemma plain_memory_unchanged_memory_write_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of unchanged_memory_write_invariant?,
Instantiating quantified variables,
Case splitting on pm_phy‘rw_addr(d!1),

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.4.1:`

```
{-1}  pm_phy`rw_addr(d')
{-2}  plain_memory?(pm_phy)
{-3}  pm_phy`rw_addr(d') ⊃
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                    (pm_phy`rw_addr \ {d'}))
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_write_transformers(pm_phy`mem, pm_phy`rw_addr),
                                  pm_phy`ro_addr)
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states = pm_phy`states
{-7}  data(linear_resolve(a', Write)(s')) = d'
{-8}  Mem?(type_of(a'))
{-9}  0 ≤ a'`offset
{-10} a'`type_of = a'`type_of
{-11} a'`offset < max_linear_offset
{-12} b' < max_byte
{-13} pm'`rw_addr(a')
{-14} pm'`states(s')
------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              (pm'`ro_addr),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Simplifying, rewriting, and recording with decision procedures,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.4.1.1:`

```
{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
                                  pm_phy‘ro_addr)
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  pm_phy‘ro_addr)
{-3}  pm_phy‘rw_addr(d′)
{-4}  plain_memory?(pm_phy)
{-5}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  (pm_phy‘rw_addr \ {d′}))
{-6}  is_linear_plain_memory?(pm′)
{-7}  pm′‘states = pm_phy‘states
{-8}  data(linear_resolve(a′, Write)(s′)) = d′
{-9}  Mem?(type_of(a′))
{-10} 0 ≤ a′‘offset
{-11} a′‘type_of = a′‘type_of
{-12} a′‘offset < max_linear_offset
{-13} b′ < max_byte
{-14} pm′‘rw_addr(a′)
{-15} pm′‘states(s′)
------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s′,
                                                          restrict
                                                              [Address, Mem-
ory_Address_4G, boolean]
                                                                  (pm′‘ro_addr),
                                                          (singleton(Read) ∪ singleton(Ex
```

Hiding formulas: -1,
Using lemma unchanged_memory_invariant_union_addresses,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2),
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 2),
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 -4 -5 -6),
Installing automatic rewrites from: subset? member union remove
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Instantiating quantified variables,
we get 2 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.4.1.1.1:`

{-1} virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, (singleton(Read) ∪ singleton(Execute))) (x')
⇒ restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x')

{-2} ∀ (a_1, a_2: Memory_Address_4G):
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_1) ∧
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr)(a_2) ∨
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_2))
⊃ ¬ linear_shared?(s')(a_1, a_2)

{-3} IF Mem?(x'‘type_of) ∧ 0 ≤ x'‘offset ∧ x'‘offset < max_linear_offset
THEN virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr), (singleton(Read) ∪ singleton(Execute))) (x')
ELSE FALSE
ENDIF

{-4} pm_phy‘rw_addr(d')
{-5} plain_memory?(pm_phy)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} data(linear_resolve(a', Write)(s')) = d'
{-9} Mem?(type_of(a'))
{-10} 0 ≤ a'‘offset
{-11} a'‘type_of = a'‘type_of
{-12} a'‘offset < max_linear_offset
{-13} b' < max_byte
{-14} pm′‘rw_addr(a')
{-15} pm′‘states(s')

{1} pm_phy‘ro_addr(x')
{2} d' ≠ x' ∧ pm_phy‘rw_addr(x')

Expanding the definition of restrict,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.1:`

{-1}	pm_phy‘rw_addr(x')
{-2}	$\forall$ (a_1, a_2: Memory_Address_4G): pm′‘rw_addr(a_1) $\wedge$ (pm′‘ro_addr(a_2) $\vee$ pm′‘rw_addr(a_2)) $\supset$ $\neg$ linear_shared?(s')(a_1, a_2)
{-3}	Mem?(x'‘type_of)
{-4}	$0 \leq x'$‘offset
{-5}	x'‘offset $<$ max_linear_offset
{-6}	virt_to_phys_range(s', λ (s: Memory_Address_4G): pm′‘ro_addr(s), (singleton(Read) $\cup$ singleton(Execute))) (x')
{-7}	pm_phy‘rw_addr(d')
{-8}	plain_memory?(pm_phy)
{-9}	is_linear_plain_memory?(pm′)
{-10}	pm′‘states = pm_phy‘states
{-11}	data(linear_resolve(a', Write)(s')) = d'
{-12}	Mem?(type_of(a'))
{-13}	$0 \leq a'$‘offset
{-14}	a'‘type_of = a'‘type_of
{-15}	a'‘offset $<$ max_linear_offset
{-16}	$b' <$ max_byte
{-17}	pm′‘rw_addr(a')
{-18}	pm′‘states(s')
{1}	pm_phy‘ro_addr(x')
{2}	$d' \neq x'$

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -7 with the terms: (a!1 lin_a!1),

Expanding the definition of linear_shared?,

Instantiating the top quantifier in 2 with the terms: (Write ac!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.1`.

`linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.2:`

{-1} $\forall$ (a_1, a_2: Memory_Address_4G):
pm′‘rw_addr(a_1) $\wedge$ (pm′‘ro_addr(a_2) $\vee$ pm′‘rw_addr(a_2)) $\supset$
$\neg$ linear_shared?(s')(a_1, a_2)
{-2} Mem?(x'‘type_of)
{-3} $0 \leq x'$‘offset
{-4} x'‘offset $<$ max_linear_offset
{-5} virt_to_phys_range(s', λ (s: Memory_Address_4G): pm′‘ro_addr(s),
(singleton(Read) $\cup$ singleton(Execute)))
(x')
{-6} pm_phy‘rw_addr(d')
{-7} plain_memory?(pm_phy)
{-8} is_linear_plain_memory?(pm′)
{-9} pm′‘states = pm_phy‘states
{-10} data(linear_resolve(a', Write)(s')) = d'
{-11} Mem?(type_of(a'))
{-12} $0 \leq a'$‘offset
{-13} a'‘type_of = a'‘type_of
{-14} a'‘offset $<$ max_linear_offset
{-15} $b' <$ max_byte
{-16} pm′‘rw_addr(a')
{-17} pm′‘states(s')

{1} virt_to_phys_range(s',
(λ (s: Memory_Address_4G): pm′‘ro_addr(s) $\cup$ λ (s: Memory_Address_4G): pm
(singleton(Read) $\cup$ singleton(Execute)))
(x')
{2} pm_phy‘ro_addr(x')
{3} pm_phy‘rw_addr(x')

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.2`.

`linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.3:`

```
{-1}  ∀ (a1, a2: Memory_Address_4G):
        pm'‘rw_addr(a1) ∧ (pm'‘ro_addr(a2) ∨ pm'‘rw_addr(a2)) ⊃
          ¬ linear_shared?(s')(a1, a2)
{-2}  Mem?(x'‘type_of)
{-3}  0 ≤ x'‘offset
{-4}  x'‘offset < max_linear_offset
{-5}  virt_to_phys_range(s', λ (s: Memory_Address_4G): pm'‘ro_addr(s),
                                 (singleton(Read) ∪ singleton(Execute)))
                               (x')
{-6}  pm_phy‘rw_addr(d')
{-7}  plain_memory?(pm_phy)
{-8}  is_linear_plain_memory?(pm')
{-9}  pm'‘states = pm_phy‘states
{-10} data(linear_resolve(a', Write)(s')) = d'
{-11} Mem?(type_of(a'))
{-12} 0 ≤ a'‘offset
{-13} a'‘type_of = a'‘type_of
{-14} a'‘offset < max_linear_offset
{-15} b' < max_byte
{-16} pm'‘rw_addr(a')
{-17} pm'‘states(s')
-------------------------------------------------------------------------------
 {1}  virt_to_phys_range(s',
                                 (λ (s : Memory_Address_4G) : pm'‘ro_addr(s) ∪ λ (s : Memory_Addres
                                 (singleton(Read) ∪ singleton(Execute)))
                               (x')
 {2}  pm_phy‘ro_addr(x')
 {3}  d' ≠ x'
```

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.1.1.1.3`.

`linear_plain_unchanged_memory_invariant_write.2.4.1.1.2:`

{-1} $\forall$ (a_1, a_2: Memory_Address_4G):
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_1) $\wedge$
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr)(a_2) $\vee$
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_2))
$\supset$ $\neg$ linear_shared?(s')(a_1, a_2)

{-2} IF Mem?(x′‘type_of) $\wedge$ $0 \leq x$′‘offset $\wedge$ x′‘offset $<$ max_linear_offset
THEN virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
(singleton(Read) $\cup$ singleton(Execute)))
(x')
ELSE FALSE
ENDIF

{-3} pm_phy‘rw_addr(d')

{-4} plain_memory?(pm_phy)

{-5} is_linear_plain_memory?(pm′)

{-6} pm′‘states = pm_phy‘states

{-7} data(linear_resolve(a', Write)(s')) = d'

{-8} Mem?(type_of(a'))

{-9} $0 \leq a$′‘offset

{-10} a′‘type_of = a′‘type_of

{-11} a′‘offset $<$ max_linear_offset

{-12} $b' <$ max_byte

{-13} pm′‘rw_addr(a')

{-14} pm′‘states(s')

{1} Mem?(x′‘type_of) $\wedge$ $0 \leq x$′‘offset $\wedge$ x′‘offset $<$ max_linear_offset

{2} pm_phy‘ro_addr(x')

{3} $d' \neq x' \wedge$ pm_phy‘rw_addr(x')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.1.1.2`.

`linear_plain_unchanged_memory_invariant_write.2.4.1.2:`

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
pm_phy‘ro_addr)
{-2} pm_phy‘rw_addr(d')
{-3} plain_memory?(pm_phy)
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))),
(pm_phy‘rw_addr \ {d'}))
{-5} is_linear_plain_memory?(pm$'$)
{-6} pm$'$‘states = pm_phy‘states
{-7} data(linear_resolve(a', Write)(s')) = d'
{-8} Mem?(type_of(a'))
{-9} 0 ≤ a'‘offset
{-10} a'‘type_of = a'‘type_of
{-11} a'‘offset < max_linear_offset
{-12} b' < max_byte
{-13} pm$'$‘rw_addr(a')
{-14} pm$'$‘states(s')

{1} (singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))) ⊆ memory_write_transformers(pm_
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Memory_Address_4G, boolean]
(pm$'$‘ro_addr),
(singleton(Read) ∪ singleton(Ex

Keeping (-2 1) and hiding *,

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of singleton,

Replacing using formula -1,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.1.2`.

`linear_plain_unchanged_memory_invariant_write.2.4.2:`

```
{-1}  plain_memory?(pm_phy)
{-2}  pm_phy`rw_addr(d') ⊃
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                    (pm_phy`rw_addr \ {d'}))
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  memory_write_transformers(pm_phy`mem, pm_phy`rw_addr),
                                  pm_phy`ro_addr)
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'`states = pm_phy`states
{-6}  data(linear_resolve(a', Write)(s')) = d'
{-7}  Mem?(type_of(a'))
{-8}  0 ≤ a'`offset
{-9}  a'`type_of = a'`type_of
{-10} a'`offset < max_linear_offset
{-11} b' < max_byte
{-12} pm'`rw_addr(a')
{-13} pm'`states(s')
------------------------------------------------------------------------------
 {1}  pm_phy`rw_addr(d')
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s',
                                                          restrict
                                                              [Address, Memory_Address_4G, boolean]
                                                              (pm'`ro_addr),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: (-2 -3 2),
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -4 -11 -17 1) and hiding *,
Expanding the definition of subset?,
Instantiating quantified variables,
Instantiating the top quantifier in -2 with the terms: d',
we get 2 subgoals:

`linear_plain_unchanged_memory_invariant_write.2.4.2.1:`

```
{-1}  restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr)(a') ⊃
        OK?(linear_resolve(a', Write)(s'))
{-2}  (d' ∈ virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean](pm'`rw_addr), singleton(Writ
        ⇒ (d' ∈ restrict[Address, Memory_Address_4G, boolean](pm_phy`rw_addr))
{-3}  data(linear_resolve(a', Write)(s')) = d'
{-4}  pm'`rw_addr(a')
------------------------------------------------------------------------------
 {1}  pm_phy`rw_addr(d')
```

Expanding the definition of member,
Expanding the definition of restrict,
Expanding the definition of virt_to_phys_range,
Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.2.1`.

`linear_plain_unchanged_memory_invariant_write.2.4.2.2:`

{-1}	restrict[Address, Memory_Address_4G, boolean](pm'rw_addr)(a') ⊃ OK?(linear_resolve(a', Write)(s'))
{-2}	data(linear_resolve(a', Write)(s')) = d'
{-3}	pm'rw_addr(a')
{1}	Mem?(d''type_of) ∧ 0 ≤ d''offset ∧ d''offset < max_linear_offset
{2}	pm_phy'rw_addr(d')

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.2.4.2.2`.

`linear_plain_unchanged_memory_invariant_write.3:`

{-1}	is_linear_plain_memory?(pm')
{1}	(restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ⊆ restrict [Address, Memory_Ad
{2}	unchanged_memory_invariant?(pm'mem, pm'states, memory_write_transformers(pm'mem, pm'rw_addr), pm'ro

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_invariant_write.3`.

Q.E.D.

C.116.4 Linear_Memory_Blessing_Properties.linear_plain_unchanged_memory_write_invar

Terse proof for `linear_plain_unchanged_memory_write_invariant`.

`linear_plain_unchanged_memory_write_invariant:`

{1}	∀ (pm: Plain_Memory[Physical_memory]): is_linear_plain_memory?(pm) ⊃ unchanged_memory_write_invariant?(pm'mem, pm'states, pm'rw_addr)

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##! has_next_state! singleton! max_linear! min_linear! Address_Helpers.<=! reg_size! reg_base! Mem! <! pm_phy_read_after_resolve_ok! pm_resolve_address! union_right! in_memory! subset_reflexive!)

Expanding the definition of unchanged_memory_write_invariant?,

Repeatedly Skolemizing and flattening,

Using lemma linear_unchanged_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_unchanged_memory_write_invariant.1:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Memory_Address_4G, boolean]
                                                ((pm'`rw_addr \ {waddr'}))))
{-3}  b' < max_byte
{-4}  pm'`rw_addr(waddr')
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  (pm'`rw_addr \ {waddr'}))
```

Case splitting on extend[Address, Memory_Address_4G, bool, FALSE] (restrict[Address, Memory_Address_4G, boolean] (remove(waddr!1, pm!1`rw_addr))) = remove(waddr!1, pm!1`rw_addr),
we get 2 subgoals:
`linear_plain_unchanged_memory_write_invariant.1.1:`

```
{-1}  extend[Address, Memory_Address_4G, bool, FALSE]
          (restrict[Address, Memory_Address_4G, boolean]((pm'`rw_addr \ {waddr'})))
       = (pm'`rw_addr \ {waddr'})
{-2}  is_linear_plain_memory?(pm')
{-3}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Memory_Address_4G, boolean]
                                                ((pm'`rw_addr \ {waddr'}))))
{-4}  b' < max_byte
{-5}  pm'`rw_addr(waddr')
------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  (pm'`rw_addr \ {waddr'}))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.1.1`.
`linear_plain_unchanged_memory_write_invariant.1.2:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (restrict[Address, Memory_Address_4G, boolean]
                                                ((pm'`rw_addr \ {waddr'}))))
{-3}  b' < max_byte
{-4}  pm'`rw_addr(waddr')
------------------------------------------------------------------------------
{1}   extend[Address, Memory_Address_4G, bool, FALSE]
          (restrict[Address, Memory_Address_4G, boolean]((pm'`rw_addr \ {waddr'})))
       = (pm'`rw_addr \ {waddr'})
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  (pm'`rw_addr \ {waddr'}))
```

Applying decompose-equality,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.1.2`.

`linear_plain_unchanged_memory_write_invariant.2`:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  b′ < max_byte
{-3}  pm′`rw_addr(waddr′)
------------------------------------------------------------------------
{1}   ∀ (s: (pm′`states)):
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_write(pm′`mem)
                                                                       (waddr′, b′))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                          (virt_to_phys_range(s,
                                                              restrict
                                                                  [Address, Mem-
ory_Address_4G, boolean]
                                                                  ((pm′`rw_addr \ {waddr′}
                                                              (singleton(Read) ∪ singleton(E
{2}   unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(memory_write(pm′`mem)(waddr′, b′))),
                                  (pm′`rw_addr \ {waddr′}))
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using pm_write_linear, matching in *,

Using lemma pm_plain_phy,

Using lemma pm_states,

Expanding the definition of linear_write,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using plain_memory_unchanged_composition[Physical_memory, Address, Unit], matching in * where pm gets pm_phy, addresses gets extend[Address, Memory_Address_4G, bool, FALSE] (virt_to_phys_range(s!1, restrict[Address, Memory_Address_4G, boolean] (remove(waddr!1, pm!1`rw_addr)), union(singleton(Read), singleton(Execute)))), P gets LAMBDA (a: Address): FORALL (s: (pm_phy`states)): OK?(linear_resolve(waddr!1, Write)(s)) IMPLIES data(linear_resolve(waddr!1, Write)(s)) = a,

we get 4 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  Mem?(type_of(waddr′))
{-3}  0 ≤ waddr′`offset
{-4}  waddr′`type_of = waddr′`type_of
{-5}  waddr′`offset < max_linear_offset
{-6}  pm′`states = pm_phy`states
{-7}  plain_memory?(pm_phy)
{-8}  pm′`states(s′)
{-9}  b′ < max_byte
{-10} pm′`rw_addr(waddr′)
------------------------------------------------------------------------------
 {1}  (extend [Address, Memory_Address_4G, bool, FALSE](virt_to_phys_range(s′, restrict [Address, Memory_
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(linear_resolve(waddr′, Write) ##
                                                         (λ (pa: Address):
                                                            mem-
      ory_write(pm_phy`mem)
                                                                (pa, b′)))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s′,
                                                        restrict
                                                          [Address, Mem-
      ory_Address_4G, boolean]
                                                          ((pm′`rw_addr \ {waddr′})),
                                                        (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: 2,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 -8 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Expanding the definition of member,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.1.1:`

```
{-1}  restrict[Address, Memory_Address_4G, boolean]((pm_phy`ro_addr ∪ pm_phy`rw_addr))(x′)
{-2}  Mem?(x′`type_of)
{-3}  0 ≤ x′`offset
{-4}  x′`offset < max_linear_offset
{-5}  virt_to_phys_range(s′,
                         restrict[Address, Memory_Address_4G, boolean]
                           ((pm′`rw_addr \ {waddr′})),
                         (singleton(Read) ∪ singleton(Execute)))
                        (x′)
{-6}  is_linear_plain_memory?(pm′)
------------------------------------------------------------------------------
 {1}  union(pm_phy`ro_addr, pm_phy`rw_addr)(x′)
```

Expanding the definition of restrict,
which is trivially true.
This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.1.1`.
`linear_plain_unchanged_memory_write_invariant.2.1.2:`

```
{-1}  Mem?(x'`type_of)
{-2}  0 ≤ x'`offset
{-3}  x'`offset < max_linear_offset
{-4}  virt_to_phys_range(s',
                         restrict[Address, Memory_Address_4G, boolean]
                             ((pm'`rw_addr \ {waddr'})),
                         (singleton(Read) ∪ singleton(Execute)))
                        (x')
{-5}  is_linear_plain_memory?(pm')
------------------------------------------------------------------------
{1}   virt_to_phys_range(s',
                         (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr) ∪ restrict
                         (singleton(Read) ∪ singleton(Execute)))
                        (x')
{2}   union(pm_phy`ro_addr, pm_phy`rw_addr)(x')
```

Keeping (-4 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:
`linear_plain_unchanged_memory_write_invariant.2.1.2.1:`

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 ≤ lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  restrict[Address, Memory_Address_4G, boolean]((pm'`rw_addr \ {waddr'}))(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  OK?(linear_resolve(lin_a', ac')(s'))
{-7}  data(linear_resolve(lin_a', ac')(s')) = x'
------------------------------------------------------------------------
{1}   OK?(linear_resolve(lin_a', ac')(s')) ∧
       data(linear_resolve(lin_a', ac')(s')) = x'
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.1.2.1`.
`linear_plain_unchanged_memory_write_invariant.2.1.2.2:`

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 ≤ lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  restrict[Address, Memory_Address_4G, boolean]((pm'`rw_addr \ {waddr'}))(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  OK?(linear_resolve(lin_a', ac')(s'))
{-7}  data(linear_resolve(lin_a', ac')(s')) = x'
------------------------------------------------------------------------
{1}   union[Memory_Address_4G]
            (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
              restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
            (lin_a')
```

Keeping (-4 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.1.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.2`:

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  Mem?(type_of(waddr′))
  {-3}  0 ≤ waddr′`offset
  {-4}  waddr′`type_of = waddr′`type_of
  {-5}  waddr′`offset < max_linear_offset
  {-6}  pm′`states = pm_phy`states
  {-7}  plain_memory?(pm_phy)
  {-8}  pm′`states(s′)
  {-9}  b′ < max_byte
 {-10}  pm′`rw_addr(waddr′)
--------------------------------------------------------------------------------
  {1}   ∀ (s_1: (pm_phy`states)):
          OK?(linear_resolve(waddr′, Write)(s_1)) ⊃
           (∀ (s: (pm_phy`states)):
              OK?(linear_resolve(waddr′, Write)(s)) ⊃
               data(linear_resolve(waddr′, Write)(s)) =
                data(linear_resolve(waddr′, Write)(s_1)))
  {2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(linear_resolve(waddr′, Write) ##
                                                           (λ (pa: Address):
                                                              mem-
        ory_write(pm_phy`mem)
                                                                   (pa, b′)))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s′,
                                                            restrict
                                                              [Address, Mem-
        ory_Address_4G, boolean]
                                                                   ((pm′`rw_addr \ {waddr′})),
                                                            (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma linear_resolve_same_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.3:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  Mem?(type_of(waddr′))
{-3}  0 ≤ waddr′‘offset
{-4}  waddr′‘type_of = waddr′‘type_of
{-5}  waddr′‘offset < max_linear_offset
{-6}  pm′‘states = pm_phy‘states
{-7}  plain_memory?(pm_phy)
{-8}  pm′‘states(s′)
{-9}  b′ < max_byte
{-10} pm′‘rw_addr(waddr′)
------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(linear_resolve(waddr′, Write))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s′,
                                                        restrict
                                                          [Address, Memory_Address_4G, boolean]
                                                          ((pm′‘rw_addr \ {waddr′}))
                                                        (singleton(Read) ∪ singleton(Ex
 {2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(linear_resolve(waddr′, Write) ##
                                                         (λ (pa: Address):
                                                            memory_write(pm_phy‘mem)
                                                              (pa, b′)))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                    (virt_to_phys_range(s′,
                                                        restrict
                                                          [Address, Memory_Address_4G, boolean]
                                                          ((pm′‘rw_addr \ {waddr′}))
                                                        (singleton(Read) ∪ singleton(Ex
```

Hiding formulas: 2,
Using lemma linear_resolve_unchanged_pm_phy,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -7,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 2),
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 -6 1) and hiding *,
Installing automatic rewrites from: (empty?! member! subset?! extend! difference! disjoint?! intersection!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1:`

```
{-1}  virt_to_phys_range(s',
                         (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address,
                         (singleton(Read) ∪ singleton(Execute)))
                        (x')
      ⇒ restrict[Address, Memory_Address_4G, boolean]((pm_phy'ro_addr ∪ pm_phy'rw_addr))(x')
{-2}  ∀ (x: Memory_Address_4G):
        ¬ (virt_to_phys_range(s',
                              (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [
                             (x)
              ∧
              address_in_pt_range?(s',
                                   (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restric
                                  (x))
{-3}  IF Mem?(x'`type_of) ∧ 0 ≤ x'`offset ∧ x'`offset < max_linear_offset
        THEN virt_to_phys_range(s',
                                restrict[Address, Memory_Address_4G, boolean]
                                    ((pm'rw_addr \ {waddr'})),
                                (singleton(Read) ∪ singleton(Execute)))
                               (x')
      ELSE FALSE
      ENDIF
-------------------------------------------------------------------------------
{1}   union(pm_phy'ro_addr, pm_phy'rw_addr)(x') ∧
       ¬ IF Mem?(x'`type_of) ∧ 0 ≤ x'`offset ∧ x'`offset < max_linear_offset
            THEN address_in_pt_range?(s',
                                      restrict[Address, Memory_Address_4G, boolean]
                                          ((pm'ro_addr ∪ pm'rw_addr)))
                                     (x')
          ELSE FALSE
          ENDIF
```

Expanding the definition of restrict,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1:`

```
{-1}  union(pm_phy'ro_addr, pm_phy'rw_addr)(x')
{-2}  ∀ (x: Memory_Address_4G):
        ¬ (virt_to_phys_range(s',
                              (λ (s: Memory_Address_4G): pm'ro_addr(s) ∪ λ (s: Memory_Address_4G
                             (x)
              ∧
              address_in_pt_range?(s',
                                   (λ (s: Memory_Address_4G): pm'ro_addr(s) ∪ λ (s: Memory_Address
                                  (x))
{-3}  Mem?(x'`type_of)
{-4}  0 ≤ x'`offset
{-5}  x'`offset < max_linear_offset
{-6}  virt_to_phys_range(s', λ (s: Memory_Address_4G): remove(waddr', pm'rw_addr)(s),
                         (singleton(Read) ∪ singleton(Execute)))
                        (x')
{-7}  address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm'ro_addr, pm'rw_addr)(s))
                           (x')
-------------------------------------------------------------------------------
```

Instantiating quantified variables,

Applying propositional simplification,

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.1:`

{-1} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x')
{-2} Mem?(x'‘type_of)
{-3} $0 \leq x'$‘offset
{-4} x'‘offset $<$ max_linear_offset
{-5} virt_to_phys_range(s', λ (s: Memory_Address_4G): remove(waddr′, pm′‘rw_addr)(s), (singleton(Read) ∪ singleton(Execute)))
(x')
{-6} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm′‘ro_addr, pm′‘rw_addr)(s))
(x')

{1} virt_to_phys_range(s',
(λ (s : Memory_Address_4G) : pm′‘ro_addr(s) ∪ λ (s : Memory_Address
(x')

Expanding the definition of virt_to_phys_range,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 3 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.1:`

{-1} Mem?(lin_a′‘type_of)
{-2} $0 \leq$ lin_a′‘offset
{-3} lin_a′‘offset $<$ max_linear_offset
{-4} remove(waddr′, pm′‘rw_addr)(lin_a′)
{-5} union(singleton(Read), singleton(Execute))(ac′)
{-6} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x')
{-7} Mem?(x'‘type_of)
{-8} $0 \leq x'$‘offset
{-9} x'‘offset $<$ max_linear_offset
{-10} OK?(linear_resolve(lin_a′, ac′)(s'))
{-11} data(linear_resolve(lin_a′, ac′)(s')) $= x'$
{-12} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm′‘ro_addr, pm′‘rw_addr)(s))
(x')

{1} OK?(linear_resolve(lin_a′, ac′)(s')) ∧
data(linear_resolve(lin_a′, ac′)(s')) $= x'$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.1`.

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.2:`

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 ≤ lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  remove(waddr', pm'`rw_addr)(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  union(pm_phy`ro_addr, pm_phy`rw_addr)(x')
{-7}  Mem?(x'`type_of)
{-8}  0 ≤ x'`offset
{-9}  x'`offset < max_linear_offset
{-10} OK?(linear_resolve(lin_a', ac')(s'))
{-11} data(linear_resolve(lin_a', ac')(s')) = x'
{-12} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s))
                           (x')
------------------------------------------------------------------------------
{1}   fullset[Memory_access](ac')
```

Expanding the definition of fullset,

which is trivially true.

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.2`.

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.3:`

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 ≤ lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  remove(waddr', pm'`rw_addr)(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  union(pm_phy`ro_addr, pm_phy`rw_addr)(x')
{-7}  Mem?(x'`type_of)
{-8}  0 ≤ x'`offset
{-9}  x'`offset < max_linear_offset
{-10} OK?(linear_resolve(lin_a', ac')(s'))
{-11} data(linear_resolve(lin_a', ac')(s')) = x'
{-12} address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s))
                           (x')
------------------------------------------------------------------------------
{1}   union[Memory_Address_4G]
          (λ (s: Memory_Address_4G): pm'`ro_addr(s),
             λ (s: Memory_Address_4G): pm'`rw_addr(s))
          (lin_a')
```

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.1.3`.

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.2:`

```
{-1}  union(pm_phy`ro_addr, pm_phy`rw_addr)(x')
{-2}  Mem?(x'`type_of)
{-3}  0 ≤ x'`offset
{-4}  x'`offset < max_linear_offset
{-5}  virt_to_phys_range(s', λ (s: Memory_Address_4G): remove(waddr', pm'`rw_addr)(s),
                                 (singleton(Read) ∪ singleton(Execute)))
                               (x')
{-6}  address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s))
                               (x')
-------------------------------------------------------------------------------------------
{1}   address_in_pt_range?(s',
                                  (λ (s : Memory_Address_4G) : pm'`ro_addr(s) ∪ λ (s : Memory_Addre
                               (x')
```

Keeping (-6 1) and hiding *,

Case splitting on (LAMBDA (s: Memory_Address_4G): union(pm!1`ro_addr, pm!1`rw_addr)(s)) = union(LAMBDA (s: Memory_Address_4G): pm!1`ro_addr(s), LAMBDA (s: Memory_Address_4G): pm!1`rw_addr(s)),

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.1:`

```
{-1}  (λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s)) =
       (λ (s : Memory_Address_4G) : pm'`ro_addr(s) ∪ λ (s : Memory_Address_4G) : pm'`rw_addr(s)
{-2}  address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s))
                               (x')
-------------------------------------------------------------------------------------------
{1}   address_in_pt_range?(s',
                                  (λ (s : Memory_Address_4G) : pm'`ro_addr(s) ∪ λ (s : Memory_Addre
                               (x')
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.1`.

`linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.2:`

```
{-1}  address_in_pt_range?(s', λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s))
                               (x')
-------------------------------------------------------------------------------------------
{1}   (λ (s: Memory_Address_4G): union(pm'`ro_addr, pm'`rw_addr)(s)) =
       (λ (s : Memory_Address_4G) : pm'`ro_addr(s) ∪ λ (s : Memory_Address_4G) : pm'`rw_addr(s)
{2}   address_in_pt_range?(s',
                                  (λ (s : Memory_Address_4G) : pm'`ro_addr(s) ∪ λ (s : Memory_Addre
                               (x')
```

Applying decompose-equality,

Hiding formulas: (-1 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.1.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.3.1.2:`

```
{-1}  ∀ (x: Memory_Address_4G):
        ¬ (virt_to_phys_range(s',
                              (λ (s: Memory_Address_4G): pm'ro_addr(s) ∪ λ (s: Memory_Address_4G
                             (x)
             ∧
             address_in_pt_range?(s',
                                  (λ (s: Memory_Address_4G): pm'ro_addr(s) ∪ λ (s: Memory_Address
                                 (x))
{-2}  Mem?(x'`type_of)
{-3}  0 ≤ x'`offset
{-4}  x'`offset < max_linear_offset
{-5}  virt_to_phys_range(s', λ (s: Memory_Address_4G): remove(waddr', pm'`rw_addr)(s),
                           (singleton(Read) ∪ singleton(Execute)))
                         (x')
------------------------------------------------------------------------------
{1}   virt_to_phys_range(s',
                           (λ (s: Memory_Address_4G): pm'ro_addr(s) ∪ λ (s: Memory_Address_4G): pm'
                           (singleton(Read) ∪ singleton(Execute)))
                         (x')
{2}   union(pm_phy`ro_addr, pm_phy`rw_addr)(x')
```

Keeping (-5 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:
`linear_plain_unchanged_memory_write_invariant.2.3.1.2.1:`

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 ≤ lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  remove(waddr', pm'`rw_addr)(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  OK?(linear_resolve(lin_a', ac')(s'))
{-7}  data(linear_resolve(lin_a', ac')(s')) = x'
------------------------------------------------------------------------------
{1}   OK?(linear_resolve(lin_a', ac')(s')) ∧
        data(linear_resolve(lin_a', ac')(s')) = x'
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.2.1`.
`linear_plain_unchanged_memory_write_invariant.2.3.1.2.2:`

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 ≤ lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  remove(waddr', pm'`rw_addr)(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  OK?(linear_resolve(lin_a', ac')(s'))
{-7}  data(linear_resolve(lin_a', ac')(s')) = x'
------------------------------------------------------------------------------
{1}   union[Memory_Address_4G]
            (λ (s: Memory_Address_4G): pm'ro_addr(s),
               λ (s: Memory_Address_4G): pm'rw_addr(s))
            (lin_a')
```

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.1.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.3.2:`

```
{-1}  ∀ (x: Memory_Address_4G):
        ¬ (virt_to_phys_range(s',
                              (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪
                            (x)
           ∧
           address_in_pt_range?(s',
                                (restrict [Address, Memory_Address_4G, boolean](pm'ro_add
                              (x))
{-2}  IF Mem?(x''type_of) ∧ 0 ≤ x''offset ∧ x''offset < max_linear_offset
        THEN virt_to_phys_range(s',
                                restrict[Address, Memory_Address_4G, boolean]
                                    ((pm'rw_addr \ {waddr'})),
                                (singleton(Read) ∪ singleton(Execute)))
                              (x')
      ELSE FALSE
      ENDIF
-------------------------------------------------------------------------------
{1}   Mem?(x''type_of) ∧ 0 ≤ x''offset ∧ x''offset < max_linear_offset
{2}   union(pm_phy'ro_addr, pm_phy'rw_addr)(x') ∧
       ¬ IF Mem?(x''type_of) ∧ 0 ≤ x''offset ∧ x''offset < max_linear_offset
            THEN address_in_pt_range?(s',
                                      restrict[Address, Memory_Address_4G, boolean]
                                          ((pm'ro_addr ∪ pm'rw_addr)))
                                    (x')
          ELSE FALSE
          ENDIF
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.3.2`.

`linear_plain_unchanged_memory_write_invariant.2.4:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  Mem?(type_of(waddr′))
  {-3}  0 ≤ waddr′`offset
  {-4}  waddr′`type_of = waddr′`type_of
  {-5}  waddr′`offset < max_linear_offset
  {-6}  pm′`states = pm_phy`states
  {-7}  plain_memory?(pm_phy)
  {-8}  pm′`states(s′)
  {-9}  b′ < max_byte
  {-10} pm′`rw_addr(waddr′)
  ------------------------------------------------------------------------
  {1}   ∀ (d:
               (λ (a: Address):
                  ∀ (s: (pm_phy`states)):
                    OK?(linear_resolve(waddr′, Write)(s)) ⊃
                     data(linear_resolve(waddr′, Write)(s)) = a)):
          unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                      singleton(expr_2_super(memory_write(pm_phy`mem)(d, b′))),
                                      extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s′,
                                                            restrict
                                                              [Address, Mem-
        ory_Address_4G, boolean]
                                                                ((pm′`rw_addr \ {waddr′})),
                                                            (singleton(Read) ∪ singleton(Execute))))
  {2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(linear_resolve(waddr′, Write) ##
                                                           (λ (pa: Address):
                                                              mem-
        ory_write(pm_phy`mem)
                                                                (pa, b′)))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s′,
                                                          restrict
                                                            [Address, Mem-
        ory_Address_4G, boolean]
                                                              ((pm′`rw_addr \ {waddr′})),
                                                          (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: s',
Using lemma pm_linear_resolve_write_ok,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_unchanged_memory_invariant_write,
Using lemma plain_memory_unchanged_memory_write_invariant,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of unchanged_memory_write_invariant?,
Instantiating quantified variables,
Using lemma unchanged_memory_invariant_union_addresses,
Case splitting on pm_phy`rw_addr(d!1),
we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.4.1:`

```
{-1}  pm_phy‘rw_addr(d′)
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  (pm_phy‘rw_addr \ {d′}))
       ∧
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  pm_phy‘ro_addr)
       ⊃
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  ((pm_phy‘rw_addr \ {d′}) ∪ pm_phy‘ro_addr))
{-3}  pm_phy‘rw_addr(d′) ⊃
       unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  (pm_phy‘rw_addr \ {d′}))
{-4}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
                                  pm_phy‘ro_addr)
{-5}  OK?(linear_resolve(waddr′, Write)(s′))
{-6}  data(linear_resolve(waddr′, Write)(s′)) = d′
{-7}  is_linear_plain_memory?(pm′)
{-8}  Mem?(type_of(waddr′))
{-9}  0 ≤ waddr′‘offset
{-10} waddr′‘type_of = waddr′‘type_of
{-11} waddr′‘offset < max_linear_offset
{-12} pm′‘states = pm_phy‘states
{-13} plain_memory?(pm_phy)
{-14} pm′‘states(s′)
{-15} b′ < max_byte
{-16} pm′‘rw_addr(waddr′)
-------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s′,
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm′‘rw_addr \ {waddr′}))
                                                          (singleton(Read) ∪ singleton(Ex
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.4.1.1:`

```
  {-1}  pm_phy`rw_addr(d')
  {-2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                    (pm_phy`rw_addr \ {d'}))
  {-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                    ((pm_phy`rw_addr \ {d'}) ∪ pm_phy`ro_addr))
  {-4}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    memory_write_transformers(pm_phy`mem, pm_phy`rw_addr),
                                    pm_phy`ro_addr)
  {-5}  OK?(linear_resolve(waddr', Write)(s'))
  {-6}  data(linear_resolve(waddr', Write)(s')) = d'
  {-7}  is_linear_plain_memory?(pm')
  {-8}  Mem?(type_of(waddr'))
  {-9}  0 ≤ waddr'`offset
  {-10} waddr'`type_of = waddr'`type_of
  {-11} waddr'`offset < max_linear_offset
  {-12} pm'`states = pm_phy`states
  {-13} plain_memory?(pm_phy)
  {-14} pm'`states(s')
  {-15} b' < max_byte
  {-16} pm'`rw_addr(waddr')
  ------------------------------------------------------------------------------
  {1}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(memory_write(pm_phy`mem)(d', b'))),
                                    extend[Address, Memory_Address_4G, bool, FALSE]
                                        (virt_to_phys_range(s',
                                                            restrict
                                                                [Address, Memory_Address_4G, boolean]
                                                                ((pm'`rw_addr \ {waddr'})),
                                                            (singleton(Read) ∪ singleton(Execute)))))
```

Hiding formulas: (-1 -3),
Using lemma unchanged_memory_invariant_mono,
Rewriting using subset_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 2),
Hiding formulas: (-1 -5 -6 -7 -12),
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 -7 -8 -9 -10 -15 1) and hiding *,
Installing automatic rewrites from: subset? member
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.4.1.1.1:`

{-1} restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x')
{-2} ∀ (a_1, a_2: Memory_Address_4G):
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_1) ∧
union(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
(a_2)
⊃ ¬ linear_shared?(s')(a_1, a_2)
{-3} Mem?(x'‘type_of)
{-4} 0 ≤ x'‘offset
{-5} x'‘offset < max_linear_offset
{-6} virt_to_phys_range(s',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘rw_addr \ {waddr′})),
(singleton(Read) ∪ singleton(Execute)))
(x')
{-7} OK?(linear_resolve(waddr′, Write)(s'))
{-8} data(linear_resolve(waddr′, Write)(s')) = d'
{-9} is_linear_plain_memory?(pm′)

{1} union((pm_phy‘rw_addr \ {d'}), pm_phy‘ro_addr)(x')

Expanding the definition of restrict,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -7 with the terms: (waddr!1 lin_a!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.1:`

{-1} Mem?(lin_a′‘type_of)
{-2} 0 ≤ lin_a′‘offset
{-3} lin_a′‘offset < max_linear_offset
{-4} remove(waddr′, pm′‘rw_addr)(lin_a′)
{-5} union(singleton(Read), singleton(Execute))(ac′)
{-6} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x')
{-7} Mem?(x'‘type_of)
{-8} 0 ≤ x'‘offset
{-9} x'‘offset < max_linear_offset
{-10} OK?(linear_resolve(lin_a′, ac′)(s'))
{-11} data(linear_resolve(lin_a′, ac′)(s')) = x'
{-12} OK?(linear_resolve(waddr′, Write)(s'))
{-13} data(linear_resolve(waddr′, Write)(s')) = d'
{-14} is_linear_plain_memory?(pm′)

{1} linear_shared?(s')(waddr′, lin_a′)
{2} union((pm_phy‘rw_addr \ {d'}), pm_phy‘ro_addr)(x')

Expanding the definition of linear_shared?,
Instantiating the top quantifier in 1 with the terms: (Write ac!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -12,
Replacing using formula -13,
Keeping (-6 1 2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.1`.
`linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.2`:

```
{-1}  Mem?(lin_a'`type_of)
{-2}  0 <= lin_a'`offset
{-3}  lin_a'`offset < max_linear_offset
{-4}  remove(waddr', pm'`rw_addr)(lin_a')
{-5}  union(singleton(Read), singleton(Execute))(ac')
{-6}  union(pm_phy`ro_addr, pm_phy`rw_addr)(x')
{-7}  Mem?(x'`type_of)
{-8}  0 <= x'`offset
{-9}  x'`offset < max_linear_offset
{-10} OK?(linear_resolve(lin_a', ac')(s'))
{-11} data(linear_resolve(lin_a', ac')(s')) = x'
{-12} OK?(linear_resolve(waddr', Write)(s'))
{-13} data(linear_resolve(waddr', Write)(s')) = d'
{-14} is_linear_plain_memory?(pm')
------------------------------------------------------------------------
 {1}  union(λ (s: Memory_Address_4G): pm'`ro_addr(s),
              λ (s: Memory_Address_4G): pm'`rw_addr(s))
            (lin_a')
 {2}  union((pm_phy`rw_addr \ {d'}), pm_phy`ro_addr)(x')
```

Keeping (-4 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.1.2`.
`linear_plain_unchanged_memory_write_invariant.2.4.1.1.2`:

```
{-1}  ∀ (a1, a2: Memory_Address_4G):
        restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr)(a1) ∧
         union(restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
                 restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
               (a2)
         ⊃ ¬ linear_shared?(s')(a1, a2)
{-2}  Mem?(x'`type_of)
{-3}  0 <= x'`offset
{-4}  x'`offset < max_linear_offset
{-5}  virt_to_phys_range(s',
                         restrict[Address, Memory_Address_4G, boolean]
                             ((pm'`rw_addr \ {waddr'})),
                         (singleton(Read) ∪ singleton(Execute)))
                        (x')
{-6}  OK?(linear_resolve(waddr', Write)(s'))
{-7}  data(linear_resolve(waddr', Write)(s')) = d'
{-8}  is_linear_plain_memory?(pm')
------------------------------------------------------------------------
 {1}  virt_to_phys_range(s',
                         (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr) ∪ restrict [Address,
                         (singleton(Read) ∪ singleton(Execute)))
                        (x')
 {2}  union((pm_phy`rw_addr \ {d'}), pm_phy`ro_addr)(x')
```

Keeping (-5 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.1:`

{-1}	Mem?(lin_a′‘type_of)
{-2}	$0 \leq$ lin_a′‘offset
{-3}	lin_a′‘offset $<$ max_linear_offset
{-4}	restrict[Address, Memory_Address_4G, boolean]((pm′‘rw_addr \ {waddr′}))(lin_a′)
{-5}	union(singleton(Read), singleton(Execute))(ac′)
{-6}	OK?(linear_resolve(lin_a′, ac′)(s′))
{-7}	data(linear_resolve(lin_a′, ac′)(s′)) $=$ x′
{1}	OK?(linear_resolve(lin_a′, ac′)(s′)) $\wedge$ data(linear_resolve(lin_a′, ac′)(s′)) $=$ x′

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.1`.

`linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.2:`

{-1}	Mem?(lin_a′‘type_of)
{-2}	$0 \leq$ lin_a′‘offset
{-3}	lin_a′‘offset $<$ max_linear_offset
{-4}	restrict[Address, Memory_Address_4G, boolean]((pm′‘rw_addr \ {waddr′}))(lin_a′)
{-5}	union(singleton(Read), singleton(Execute))(ac′)
{-6}	OK?(linear_resolve(lin_a′, ac′)(s′))
{-7}	data(linear_resolve(lin_a′, ac′)(s′)) $=$ x′
{1}	union[Memory_Address_4G] (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)) (lin_a′)

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.1.2.2`.

`linear_plain_unchanged_memory_write_invariant.2.4.1.2:`

{-1} pm_phy‘rw_addr(d')
{-2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))),
(pm_phy‘rw_addr \ {d'}))
{-3} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
pm_phy‘ro_addr)
{-4} OK?(linear_resolve(waddr$'$, Write)(s'))
{-5} data(linear_resolve(waddr$'$, Write)(s')) = d'
{-6} is_linear_plain_memory?(pm$'$)
{-7} Mem?(type_of(waddr$'$))
{-8} 0 ≤ waddr$'$‘offset
{-9} waddr$'$‘type_of = waddr$'$‘type_of
{-10} waddr$'$‘offset < max_linear_offset
{-11} pm$'$‘states = pm_phy‘states
{-12} plain_memory?(pm_phy)
{-13} pm$'$‘states(s')
{-14} b' < max_byte
{-15} pm$'$‘rw_addr(waddr$'$)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))),
pm_phy‘ro_addr)
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(memory_write(pm_phy‘mem)(d', b'))),
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s',
restrict
[Address, Memory_Address_4G, boolean]
((pm$'$‘rw_addr \ {waddr$'$})),
(singleton(Read) ∪ singleton(Execute)))))

Hiding formulas: (-2 2),

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 2),

Expanding the definition of subset?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Expanding the definition of memory_write_transformers,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.1.2`.

`linear_plain_unchanged_memory_write_invariant.2.4.2:`

```
{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  (pm_phy‘rw_addr \ {d′}))
       ∧
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  pm_phy‘ro_addr)
       ⊃
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  ((pm_phy‘rw_addr \ {d′}) ∪ pm_phy‘ro_addr))
{-2}  pm_phy‘rw_addr(d′) ⊃
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  (pm_phy‘rw_addr \ {d′}))
{-3}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
                                  pm_phy‘ro_addr)
{-4}  OK?(linear_resolve(waddr′, Write)(s′))
{-5}  data(linear_resolve(waddr′, Write)(s′)) = d′
{-6}  is_linear_plain_memory?(pm′)
{-7}  Mem?(type_of(waddr′))
{-8}  0 ≤ waddr′‘offset
{-9}  waddr′‘type_of = waddr′‘type_of
{-10} waddr′‘offset < max_linear_offset
{-11} pm′‘states = pm_phy‘states
{-12} plain_memory?(pm_phy)
{-13} pm′‘states(s′)
{-14} b′ < max_byte
{-15} pm′‘rw_addr(waddr′)
------------------------------------------------------------------------
 {1}  pm_phy‘rw_addr(d′)
 {2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(memory_write(pm_phy‘mem)(d′, b′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE]
                                      (virt_to_phys_range(s′,
                                                          restrict
                                                              [Address, Mem-
      ory_Address_4G, boolean]
                                                              ((pm′‘rw_addr \ {waddr′})),
                                                          (singleton(Read) ∪ singleton(Ex
```

Hiding formulas: (-1 -2 -3 2),
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-4 -8 -9 -18 1) and hiding *,
Expanding the definition of subset?,
Expanding the definition of member,
Instantiating quantified variables,
we get 2 subgoals:

`linear_plain_unchanged_memory_write_invariant.2.4.2.1:`

```
{-1}  virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr),
                                 singleton(Write))
                               (d')
       ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy`rw_addr)(d')
{-2}  OK?(linear_resolve(waddr', Write)(s'))
{-3}  data(linear_resolve(waddr', Write)(s')) = d'
{-4}  b' < max_byte
------------------------------------------------------------------------
{1}   pm_phy`rw_addr(d')
```

Expanding the definition of restrict,
Expanding the definition of virt_to_phys_range,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.2.1`.
`linear_plain_unchanged_memory_write_invariant.2.4.2.2:`

```
{-1}  OK?(linear_resolve(waddr', Write)(s'))
{-2}  data(linear_resolve(waddr', Write)(s')) = d'
{-3}  b' < max_byte
------------------------------------------------------------------------
{1}   Mem?(d'`type_of) ∧ 0 ≤ d'`offset ∧ d'`offset < max_linear_offset
{2}   pm_phy`rw_addr(d')
```

Using lemma linear_resolve_memory_address,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.2.4.2.2`.
`linear_plain_unchanged_memory_write_invariant.3:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  b' < max_byte
{-3}  pm'`rw_addr(waddr')
------------------------------------------------------------------------
{1}   (restrict[Address, Memory_Address_4G, boolean]((pm'`rw_addr \ {waddr'})) ⊆ restrict [Address, Memory_A
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(memory_write(pm'`mem)(waddr', b'))),
                                  (pm'`rw_addr \ {waddr'}))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_unchanged_memory_write_invariant.3`.
Q.E.D.

C.116.5 Linear_Memory_Blessing_Properties.linear_plain_transformer_invariant

Terse proof for `linear_plain_transformer_invariant`.
`linear_plain_transformer_invariant:`

```
------------------------------------------------------------------------
{1}   ∀ (pm: Plain_Memory[Physical_memory]):
        is_linear_plain_memory?(pm) ⊃
          transformer_invariant?(pm`states,
                                 ((pm`other_actions ∪ (memory_read_transformers(pm`mem, (pm`ro_addr ∪ pm'
```

Repeatedly Skolemizing and flattening,

Using lemma linear_plain_unchanged_memory_invariant,

Using lemma unchanged_memory_invariant_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma linear_plain_unchanged_memory_invariant_write,

Using lemma unchanged_memory_invariant_invariant,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 1) and hiding *,

Using lemma transformer_invariant_union_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on union(memory_write_transformers(pm!1‘mem, pm!1‘rw_addr), union(union(pm!1‘other_actions, memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr))), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr)))) = union(union(pm!1‘other_actions, union(memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr))), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),

we get 2 subgoals:

`linear_plain_transformer_invariant.1:`

```
{-1}  (memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_actions ∪ memory_read_transf
        =
       ((pm′‘other_actions ∪ (memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ∪ mem
{-2}  transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))
{-3}  transformer_invariant?(pm′‘states,
                             ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_a
{-4}  transformer_invariant?(pm′‘states,
                             (memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_a
------------------------------------------------------------------------------------------------
{1}   transformer_invariant?(pm′‘states,
                             ((pm′‘other_actions ∪ (memory_read_transformers(pm′‘mem, (pm′‘ro_
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_transformer_invariant.1`.

`linear_plain_transformer_invariant.2:`

```
{-1}  transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))
{-2}  transformer_invariant?(pm′‘states,
                             ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_a
{-3}  transformer_invariant?(pm′‘states,
                             (memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_a
------------------------------------------------------------------------------------------------
{1}   (memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_actions ∪ memory_read_transf
        =
       ((pm′‘other_actions ∪ (memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ∪ mem
{2}   transformer_invariant?(pm′‘states,
                             ((pm′‘other_actions ∪ (memory_read_transformers(pm′‘mem, (pm′‘ro_
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_transformer_invariant.2`.

Q.E.D.

C.116.6 Linear_Memory_Blessing_Properties.linear_plain_changed_memory_invariant

Terse proof for `linear_plain_changed_memory_invariant`.

`linear_plain_changed_memory_invariant:`

```
{1}  ∀ (pm: Plain_Memory[Physical_memory]):
       is_linear_plain_memory?(pm) ⊃ changed_memory_invariant?(pm`mem, pm`states, pm`rw_addr)
```

Installing automatic rewrites from: has_next_state fatal_result singleton

Expanding the definition of changed_memory_invariant?,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_changed_memory_invariant.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{1}   ∀ (s: Physical_memory, a: Address, b: Byte):
        pm′`states(s) ∧
         pm′`rw_addr(a) ∧
          OK?(memory_write(pm′`mem)(a, b)(s)) ∧
           OK?(memory_read(pm′`mem)(a)(state(memory_write(pm′`mem)(a, b)(s))))
         ⊃
         data(memory_read(pm′`mem)(a)(state(memory_write(pm′`mem)(a, b)(s)))) = b
```

Repeatedly Skolemizing and flattening,

Rewriting using pm_read_linear, matching in *,

Rewriting using pm_write_linear, matching in *,

Expanding the definition of linear_read,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##!)

Expanding the definition of linear_write,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_changed_memory_invariant.1.1:`

```
{-1}  b′ < max_byte
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  pm′‘rw_addr(a′)
{-5}  in_memory(min_linear, max_linear)(a′)
{-6}  Mem?(type_of(a′))
{-7}  OK?(linear_resolve(a′, Write)(s′))
{-8}  OK?(memory_write(pm_phy‘mem)
                (data(linear_resolve(a′, Write)(s′)), b′)
                (state(linear_resolve(a′, Write)(s′))))
{-9}  OK?(linear_resolve(a′, Read)
                        (state(memory_write(pm_phy‘mem)
                                           (data(linear_resolve(a′, Write)(s′)), b′)
                                           (state(linear_resolve(a′, Write)(s′))))))
{-10} OK?(memory_read(pm_phy‘mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy‘mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)(s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy‘mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′))))))))
-------------------------------------------------------------------------------------------
{1}   data(memory_read(pm_phy‘mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy‘mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy‘mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′))))))))
      = b′
```

Using lemma linear_resolve_states,

Using lemma pm_plain_phy,

Using lemma pm_states,

Case splitting on union(pm!1‘ro_addr, pm!1‘rw_addr)(a!1),

we get 2 subgoals:

`linear_plain_changed_memory_invariant.1.1.1:`

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-2} is_linear_plain_memory?(pm′) ⊃ pm′‘states = pm_phy‘states
{-3} is_linear_plain_memory?(pm′) ⊃ plain_memory?(pm_phy)
{-4} union(pm′‘ro_addr, pm′‘rw_addr)(a') ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s') ∧ OK?(linear_resolve(a', Write)(s'))
⊃ pm_phy‘states(state(linear_resolve(a', Write)(s')))
{-5} b' < max_byte
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states(s')
{-8} pm′‘rw_addr(a')
{-9} in_memory(min_linear, max_linear)(a')
{-10} Mem?(type_of(a'))
{-11} OK?(linear_resolve(a', Write)(s'))
{-12} OK?(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))
{-13} OK?(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))
{-14} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

{1} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))
= b'

Using lemma super_transformer_invariant_next_ok,

Case splitting on pm_phy‘rw_addr(data(linear_resolve(a!1, Write)(s!1))),

we get 2 subgoals:

`linear_plain_changed_memory_invariant.1.1.1.1:`

{-1} pm_phy‘rw_addr(data(linear_resolve(a', Write)(s')))

{-2} transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_
pm_phy‘states(state(linear_resolve(a', Write)(s'))) ∧
memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr)
(expr_2_super(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')),
b')))
∧
has_next_state(expr_2_super(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b'))
(state(linear_resolve(a', Write)(s'))))
⊃
pm_phy‘states
(state(expr_2_super(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b'))
(state(linear_resolve(a', Write)(s')))))

{-3} union(pm′‘ro_addr, pm′‘rw_addr)(a')

{-4} is_linear_plain_memory?(pm′) ⊃ pm′‘states = pm_phy‘states

{-5} is_linear_plain_memory?(pm′) ⊃ plain_memory?(pm_phy)

{-6} union(pm′‘ro_addr, pm′‘rw_addr)(a') ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s') ∧ OK?(linear_resolve(a', Write)(s'))
⊃ pm_phy‘states(state(linear_resolve(a', Write)(s')))

{-7} b' < max_byte

{-8} is_linear_plain_memory?(pm′)

{-9} pm′‘states(s')

{-10} pm′‘rw_addr(a')

{-11} in_memory(min_linear, max_linear)(a')

{-12} Mem?(type_of(a'))

{-13} OK?(linear_resolve(a', Write)(s'))

{-14} OK?(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))

{-15} OK?(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))

{-16} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

{1} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

= b'

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_changed_memory_invariant,

Expanding the definition of changed_memory_invariant?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_changed_memory_invariant.1.1.1.1.1:`

```
{-1}  transformer_invariant?(pm_phy`states, memory_read_transformers(pm_phy`mem, pm_phy`rw_a
{-2}  transformer_invariant?(pm_phy`states, memory_write_transformers(pm_phy`mem, pm_phy`rw_
{-3}  ∀ (s: Physical_memory, a: Address, b: Byte):
        pm_phy`states(s) ∧
         pm_phy`rw_addr(a) ∧
          OK?(memory_write(pm_phy`mem)(a, b)(s)) ∧
           OK?(memory_read(pm_phy`mem)(a)(state(memory_write(pm_phy`mem)(a, b)(s))))
         ⊃
         data(memory_read(pm_phy`mem)(a)(state(memory_write(pm_phy`mem)(a, b)(s))))
          = b
{-4}  pm_phy`rw_addr(data(linear_resolve(a′, Write)(s′)))
{-5}  pm_phy`states(state(linear_resolve(a′, Write)(s′)))
{-6}  pm_phy`states
        (state(memory_write(pm_phy`mem)
                           (data(linear_resolve(a′, Write)(s′)), b′)
                           (state(linear_resolve(a′, Write)(s′)))))
{-7}  union(pm′`ro_addr, pm′`rw_addr)(a′)
{-8}  pm′`states = pm_phy`states
{-9}  plain_memory?(pm_phy)
{-10} b′ < max_byte
{-11} is_linear_plain_memory?(pm′)
{-12} pm′`states(s′)
{-13} pm′`rw_addr(a′)
{-14} in_memory(min_linear, max_linear)(a′)
{-15} Mem?(type_of(a′))
{-16} OK?(linear_resolve(a′, Write)(s′))
{-17} OK?(memory_write(pm_phy`mem)
                (data(linear_resolve(a′, Write)(s′)), b′)
                (state(linear_resolve(a′, Write)(s′))))
{-18} OK?(linear_resolve(a′, Read)
                        (state(memory_write(pm_phy`mem)
                                           (data(linear_resolve(a′, Write)(s′)), b′)
                                           (state(linear_resolve(a′, Write)(s′))))))
{-19} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)(s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′))))))))
-------------------------------------------------------------------------------------------
{1}   data(memory_read(pm_phy`mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′))))))))
      = b′
```

Instantiating the top quantifier in -3 with the terms: (state(linear_resolve(a!1, Write)(s!1)) data(linear_resolve(a!1, Write)(s!1)) b!1),

Using lemma linear_resolve_unchanged_pm_phy,

Using lemma unchanged_memory_invariant_unchanged,

Case splitting on data(linear_resolve(a!1, Read) (state(memory_write(pm_phy‘mem) (data(linear_resolve(a!1, Write)(s!1)), b!1) (state(linear_resolve (a!1, Write)(s!1)))))) = data(linear_resolve(a!1, Write)(s!1)),

we get 2 subgoals:

```
linear_plain_changed_memory_invariant.1.1.1.1.1.1:

{-1}  data(linear_resolve(a′, Read)
                (state(memory_write(pm_phy‘mem)
                                   (data(linear_resolve(a′, Write)(s′)), b′)
                                   (state(linear_resolve(a′, Write)(s′))))))
       = data(linear_resolve(a′, Write)(s′))
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
                                  singleton(expr_2_super[Physical_memory, Address]
                                                        (linear_resolve(a′, Read))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
       ∧
       pm′‘states
            (state(memory_write(pm_phy‘mem)
                               (data(linear_resolve(a′, Write)(s′)), b′)
                               (state(linear_resolve(a′, Write)(s′)))))
        ∧
        singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
                  (expr_2_super(linear_resolve(a′, Read)))
         ∧
         difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
                    extend[Address, Memory_Address_4G, bool, FALSE]
                          (address_in_pt_range?(s′,
                                                restrict[Address, Mem-
ory_Address_4G, boolean]
                                                        ((pm′‘ro_addr ∪ pm′‘rw_addr)))))
                    (data(linear_resolve(a′, Write)(s′)))
          ∧
          OK?(expr_2_super(linear_resolve(a′, Read))
                          (state(memory_write(pm_phy‘mem)
                                             (data(linear_resolve(a′, Write)(s′)), b′)
                                             (state(linear_resolve(a′, Write)(s′))))))
           ∧
           OK?(memory_read(pm_phy‘mem)
                          (data(linear_resolve(a′, Write)(s′)))
                          (state(memory_write(pm_phy‘mem)
                                             (data(linear_resolve(a′, Write)(s′)), b′)
                                             (state(linear_resolve(a′, Write)(s′))))))
            ∧
            OK?(memory_read(pm_phy‘mem)
                           (data(linear_resolve(a′, Write)(s′)))
                           (state(expr_2_super(linear_resolve(a′, Read))
                                              (state(memory_write(pm_phy‘mem)
                                                                 (data(linear_resolve(a′, Write)(s′)),
                                                                   b′)
                                                                 (state(linear_resolve(a′, Write)
                                                                                      (s′)))))))))
       ⊃
       data(memory_read(pm_phy‘mem)
                       (data(linear_resolve(a′, Write)(s′)))
                       (state(expr_2_super(linear_resolve(a′, Read))
                                          (state(memory_write(pm_phy‘mem)
                                                             (data(linear_resolve(a′, Write)(s′)), b′)
                                                             (state(linear_resolve(a′, Write)
                                                                                  (s′)))))))))
        =
        data(memory_read(pm_phy‘mem)
                        (data(linear_resolve(a′, Write)(s′)))
                        (state(memory_write(pm_phy‘mem)
                                           (data(linear_resolve(a′, Write)(s′)), b′)
                                           (state(linear_resolve(a′, Write)(s′))))))
{-3}  union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)
       ⊃
       unchanged_memory_invariant?[Physical_memory]
            (pm_phy‘mem, pm′‘states,
```

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`linear_plain_changed_memory_invariant.1.1.1.1.1.1.1:`

```
{-1}  data(linear_resolve(a′, Read)
                          (state(memory_write(pm_phy‘mem)
                                             (data(linear_resolve(a′, Write)(s′)), b′)
                                             (state(linear_resolve(a′, Write)(s′))))))
       = data(linear_resolve(a′, Write)(s′))
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
                                  singleton(expr_2_super[Physical_memory, Address]
                                                        (linear_resolve(a′, Read))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
{-3}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a
{-7}  transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_
{-8}  pm_phy‘states(state(linear_resolve(a′, Write)(s′)))
{-9}  pm_phy‘rw_addr(data(linear_resolve(a′, Write)(s′)))
{-10} OK?(memory_write(pm_phy‘mem)
                       (data(linear_resolve(a′, Write)(s′)), b′)
                       (state(linear_resolve(a′, Write)(s′))))
{-11} pm_phy‘states
          (state(memory_write(pm_phy‘mem)
                             (data(linear_resolve(a′, Write)(s′)), b′)
                             (state(linear_resolve(a′, Write)(s′)))))
{-12} pm′‘states = pm_phy‘states
{-13} plain_memory?(pm_phy)
{-14} b′ < max_byte
{-15} pm′‘rw_addr(a′)
{-16} in_memory(min_linear, max_linear)(a′)
{-17} Mem?(type_of(a′))
{-18} OK?(linear_resolve(a′, Write)(s′))
{-19} OK?(linear_resolve(a′, Read)
                         (state(memory_write(pm_phy‘mem)
                                            (data(linear_resolve(a′, Write)(s′)), b′)
                                            (state(linear_resolve(a′, Write)(s′))))))
{-20} OK?(memory_read(pm_phy‘mem)
                      (data(linear_resolve(a′, Write)(s′)))
                      (state(linear_resolve(a′, Read)
                                           (state(memory_write(pm_phy‘mem)
                                                              (data(linear_resolve(a′, Write)(s′)), b′)
                                                              (state(linear_resolve(a′, Write)
                                                                                   (s′)))))))))
-------------------------------------------------------------------------------------------------
 {1}  OK?(memory_read(pm_phy‘mem)
                      (data(linear_resolve(a′, Write)(s′)))
                      (state(memory_write(pm_phy‘mem)
                                         (data(linear_resolve(a′, Write)(s′)), b′)
                                         (state(linear_resolve(a′, Write)(s′))))))
 {2}  data(memory_read(pm_phy‘mem)
                       (data(linear_resolve(a′, Write)(s′)))
                       (state(linear_resolve(a′, Read)
                                            (state(memory_write(pm_phy‘mem)
                                                               (data(linear_resolve(a′, Write)(s′)), b′)
                                                               (state(linear_resolve(a′, Write)
                                                                                    (s′)))))))))
       = b′
```

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using memory_read_transformers_memory_read, matching in *,

Keeping (-12 1) and hiding *,

Rewriting using union_right, matching in *,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.1.1`.

linear_plain_changed_memory_invariant.1.1.1.1.1.1.2:

{-1} data(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
= data(linear_resolve(a′, Write)(s′))
{-2} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states(s′)
{-6} transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a
{-7} transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_
{-8} pm_phy‘states(state(linear_resolve(a′, Write)(s′)))
{-9} pm_phy‘rw_addr(data(linear_resolve(a′, Write)(s′)))
{-10} OK?(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))
{-11} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
= b′
{-12} pm_phy‘states
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′)))))
{-13} pm′‘states = pm_phy‘states
{-14} plain_memory?(pm_phy)
{-15} b′ < max_byte
{-16} pm′‘rw_addr(a′)
{-17} in_memory(min_linear, max_linear)(a′)
{-18} Mem?(type_of(a′))
{-19} OK?(linear_resolve(a′, Write)(s′))
{-20} OK?(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
{-21} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)
(s′)))))))

{1} difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Write)(s′)))
{2} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)
(s′)))))))
= b′

Hiding formulas: 2,

Using lemma pm_resolve_address_write,

we get 2 subgoals:

`linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.1:`

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
virt_to_phys_range(s′, restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr),
singleton(Write))
(data(linear_resolve(a′, Write)(s′)))
⊃
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Write)(s′)))

{-2} data(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
= data(linear_resolve(a′, Write)(s′))

{-3} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem

{-4} union(pm′‘ro_addr, pm′‘rw_addr)(a′)

{-5} is_linear_plain_memory?(pm′)

{-6} pm′‘states(s′)

{-7} transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a

{-8} transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_

{-9} pm_phy‘states(state(linear_resolve(a′, Write)(s′)))

{-10} pm_phy‘rw_addr(data(linear_resolve(a′, Write)(s′)))

{-11} OK?(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))

{-12} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
= b′

{-13} pm_phy‘states
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′)))))

{-14} pm′‘states = pm_phy‘states

{-15} plain_memory?(pm_phy)

{-16} b′ < max_byte

{-17} pm′‘rw_addr(a′)

{-18} in_memory(min_linear, max_linear)(a′)

{-19} Mem?(type_of(a′))

{-20} OK?(linear_resolve(a′, Write)(s′))

{-21} OK?(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))

{-22} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(linear_resolve(a′, Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)
(s′))))))))

{1} difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

Expanding the definition of singleton,

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.1`.

`linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.2:`

{-1} data(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))
= data(linear_resolve(a', Write)(s'))
{-2} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a', Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states(s')
{-6} transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a
{-7} transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_
{-8} pm_phy‘states(state(linear_resolve(a', Write)(s')))
{-9} pm_phy‘rw_addr(data(linear_resolve(a', Write)(s')))
{-10} OK?(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))
{-11} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))
= b'
{-12} pm_phy‘states
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s')))))
{-13} pm′‘states = pm_phy‘states
{-14} plain_memory?(pm_phy)
{-15} $b' <$ max_byte
{-16} pm′‘rw_addr(a')
{-17} in_memory(min_linear, max_linear)(a')
{-18} Mem?(type_of(a'))
{-19} OK?(linear_resolve(a', Write)(s'))
{-20} OK?(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))
{-21} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

{1} Mem?(data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a', Write)(s'))‘type_of)
∧
$0 \leq$
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a', Write)(s'))‘offset
∧
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a', Write)(s'))‘offset
$<$ max_linear_offset
{2} difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

which is trivially true.

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.1.2.2`.

`linear_plain_changed_memory_invariant.1.1.1.1.1.1.3:`

```
{-1}  data(linear_resolve(a′, Read)
                          (state(memory_write(pm_phy‘mem)
                                             (data(linear_resolve(a′, Write)(s′)), b′)
                                             (state(linear_resolve(a′, Write)(s′))))))
       = data(linear_resolve(a′, Write)(s′))
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
                                  singleton(expr_2_super[Physical_memory, Address]
                                                        (linear_resolve(a′, Read))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
{-3}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a
{-7}  transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_
{-8}  pm_phy‘states(state(linear_resolve(a′, Write)(s′)))
{-9}  pm_phy‘rw_addr(data(linear_resolve(a′, Write)(s′)))
{-10} OK?(memory_write(pm_phy‘mem)
                      (data(linear_resolve(a′, Write)(s′)), b′)
                      (state(linear_resolve(a′, Write)(s′))))
{-11} data(memory_read(pm_phy‘mem)
                       (data(linear_resolve(a′, Write)(s′)))
                       (state(memory_write(pm_phy‘mem)
                                          (data(linear_resolve(a′, Write)(s′)), b′)
                                          (state(linear_resolve(a′, Write)(s′))))))
       = b′
{-12} pm_phy‘states
           (state(memory_write(pm_phy‘mem)
                              (data(linear_resolve(a′, Write)(s′)), b′)
                              (state(linear_resolve(a′, Write)(s′)))))
{-13} pm′‘states = pm_phy‘states
{-14} plain_memory?(pm_phy)
{-15} b′ < max_byte
{-16} pm′‘rw_addr(a′)
{-17} in_memory(min_linear, max_linear)(a′)
{-18} Mem?(type_of(a′))
{-19} OK?(linear_resolve(a′, Write)(s′))
{-20} OK?(linear_resolve(a′, Read)
                        (state(memory_write(pm_phy‘mem)
                                           (data(linear_resolve(a′, Write)(s′)), b′)
                                           (state(linear_resolve(a′, Write)(s′))))))
{-21} OK?(memory_read(pm_phy‘mem)
                      (data(linear_resolve(a′, Write)(s′)))
                      (state(linear_resolve(a′, Read)
                                           (state(memory_write(pm_phy‘mem)
                                                              (data(linear_resolve(a′, Write)(s′)), b′)
                                                              (state(linear_resolve(a′, Write)
                                                                                   (s′)))))))))
-------------------------------------------------------------------------------------------------
{1}   expr_2_super(linear_resolve(a′, Read)) =
       expr_2_super[Physical_memory, Address](linear_resolve(a′, Read))
{2}   data(memory_read(pm_phy‘mem)
                       (data(linear_resolve(a′, Write)(s′)))
                       (state(linear_resolve(a′, Read)
                                            (state(memory_write(pm_phy‘mem)
                                                               (data(linear_resolve(a′, Write)(s′)), b′)
                                                               (state(linear_resolve(a′, Write)
                                                                                    (s′)))))))))
       = b′
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.1.3`.

linear_plain_changed_memory_invariant.1.1.1.1.1.2:

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
∧
pm′‘states
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′)))))
∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Write)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)),
b′)
(state(linear_resolve(a′, Write)
(s′)))))))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)
(s′)))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)))
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a′, Write)(s′)), b′)
(state(linear_resolve(a′, Write)(s′))))))

{-2} union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)
⊃
unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, FA

{-3} transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a

{-4} transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_

{-5} pm_phy‘states(state(linear_resolve(a′, Write)(s′))) ∧

Hiding formulas: 2,

Using lemma linear_resolve_same_result,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.1.2`.

`linear_plain_changed_memory_invariant.1.1.1.1.2:`

```
{-1}  transformer_invariant?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, pm_phy‘rw_a
{-2}  transformer_invariant?(pm_phy‘states, memory_write_transformers(pm_phy‘mem, pm_phy‘rw_
{-3}  ∀ (s: Physical_memory, a: Address, b: Byte):
        pm_phy‘states(s) ∧
         pm_phy‘rw_addr(a) ∧
          OK?(memory_write(pm_phy‘mem)(a, b)(s)) ∧
           OK?(memory_read(pm_phy‘mem)(a)(state(memory_write(pm_phy‘mem)(a, b)(s))))
         ⊃
         data(memory_read(pm_phy‘mem)(a)(state(memory_write(pm_phy‘mem)(a, b)(s))))
          = b
{-4}  pm_phy‘rw_addr(data(linear_resolve(a′, Write)(s′)))
{-5}  pm_phy‘states(state(linear_resolve(a′, Write)(s′)))
{-6}  union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-7}  pm′‘states = pm_phy‘states
{-8}  plain_memory?(pm_phy)
{-9}  b′ < max_byte
{-10} is_linear_plain_memory?(pm′)
{-11} pm′‘states(s′)
{-12} pm′‘rw_addr(a′)
{-13} in_memory(min_linear, max_linear)(a′)
{-14} Mem?(type_of(a′))
{-15} OK?(linear_resolve(a′, Write)(s′))
{-16} OK?(memory_write(pm_phy‘mem)
                (data(linear_resolve(a′, Write)(s′)), b′)
                (state(linear_resolve(a′, Write)(s′))))
{-17} OK?(linear_resolve(a′, Read)
                        (state(memory_write(pm_phy‘mem)
                                           (data(linear_resolve(a′, Write)(s′)), b′)
                                           (state(linear_resolve(a′, Write)(s′))))))
{-18} OK?(memory_read(pm_phy‘mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy‘mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)(s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy‘mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′)))))))
-------------------------------------------------------------------------------------
{1}   memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr)
                               (expr_2_super(memory_write(pm_phy‘mem)
                                                         (data(linear_resolve(a′, Write)(s′)),
                                                          b′)))
{2}   data(memory_read(pm_phy‘mem)
                      (data(linear_resolve(a′, Read)
                                          (state(memory_write(pm_phy‘mem)
                                                             (data(linear_resolve(a′, Write)(s′)), b′)
                                                             (state(linear_resolve(a′, Write)
                                                                                  (s′)))))))
                      (state(linear_resolve(a′, Read)
                                           (state(memory_write(pm_phy‘mem)
                                                              (data(linear_resolve(a′, Write)(s′)), b′)
                                                              (state(linear_resolve(a′, Write)
                                                                                   (s′)))))))
        = b′
```

Rewriting using memory_write_transformers_memory_write, matching in *,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.1.2`.

`linear_plain_changed_memory_invariant.1.1.1.2:`

{-1} transformer_invariant?(pm_phy`states, memory_write_transformers(pm_phy`mem, pm_phy`rw_
pm_phy`states(state(linear_resolve(a', Write)(s'))) ∧
memory_write_transformers(pm_phy`mem, pm_phy`rw_addr)
(expr_2_super(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')),
b')))
∧
has_next_state(expr_2_super(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b'))
(state(linear_resolve(a', Write)(s'))))
⊃
pm_phy`states
(state(expr_2_super(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b'))
(state(linear_resolve(a', Write)(s')))))

{-2} union(pm′`ro_addr, pm′`rw_addr)(a')

{-3} is_linear_plain_memory?(pm′) ⊃ pm′`states = pm_phy`states

{-4} is_linear_plain_memory?(pm′) ⊃ plain_memory?(pm_phy)

{-5} union(pm′`ro_addr, pm′`rw_addr)(a') ∧
is_linear_plain_memory?(pm′) ∧
pm′`states(s') ∧ OK?(linear_resolve(a', Write)(s'))
⊃ pm_phy`states(state(linear_resolve(a', Write)(s')))

{-6} b' < max_byte

{-7} is_linear_plain_memory?(pm′)

{-8} pm′`states(s')

{-9} pm′`rw_addr(a')

{-10} in_memory(min_linear, max_linear)(a')

{-11} Mem?(type_of(a'))

{-12} OK?(linear_resolve(a', Write)(s'))

{-13} OK?(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))

{-14} OK?(linear_resolve(a', Read)
(state(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))

{-15} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

{1} pm_phy`rw_addr(data(linear_resolve(a', Write)(s')))

{2} data(memory_read(pm_phy`mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy`mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

= b'

Hiding formulas: 2,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Hiding formulas: (-2 -6),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -5 -6 -7),

Expanding the definition of subset?,

Expanding the definition of member,

Expanding the definition of restrict,

Instantiating quantified variables,

we get 2 subgoals:

`linear_plain_changed_memory_invariant.1.1.1.2.1:`

```
{-1}  virt_to_phys_range(s', λ (s: Memory_Address_4G): pm'`rw_addr(s), singleton(Write))
                           (data(linear_resolve(a', Write)(s')))
       ⇒ pm_phy`rw_addr(data(linear_resolve(a', Write)(s')))
{-2}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-3}  is_linear_plain_memory?(pm')
{-4}  pm'`states = pm_phy`states
{-5}  plain_memory?(pm_phy)
{-6}  b' < max_byte
{-7}  pm'`states(s')
{-8}  pm'`rw_addr(a')
{-9}  in_memory(min_linear, max_linear)(a')
{-10} Mem?(type_of(a'))
{-11} OK?(linear_resolve(a', Write)(s'))
{-12} OK?(memory_write(pm_phy`mem)
                (data(linear_resolve(a', Write)(s')), b')
                (state(linear_resolve(a', Write)(s'))))
{-13} OK?(linear_resolve(a', Read)
                          (state(memory_write(pm_phy`mem)
                                             (data(linear_resolve(a', Write)(s')), b')
                                             (state(linear_resolve(a', Write)(s'))))))
{-14} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)
                                          (state(memory_write(pm_phy`mem)
                                                             (data(linear_resolve(a', Write)(s')), b')
                                                             (state(linear_resolve(a', Write)(s')))))))
                (state(linear_resolve(a', Read)
                                          (state(memory_write(pm_phy`mem)
                                                             (data(linear_resolve(a', Write)(s')), b')
                                                             (state(linear_resolve(a', Write)
                                                                                  (s'))))))))
-------------------------------------------------------------------------------------------------
 {1}  pm_phy`rw_addr(data(linear_resolve(a', Write)(s')))
```

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.2.1`.

`linear_plain_changed_memory_invariant.1.1.1.2.2:`

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-2} is_linear_plain_memory?(pm′)
{-3} pm′‘states = pm_phy‘states
{-4} plain_memory?(pm_phy)
{-5} $b' <$ max_byte
{-6} pm′‘states(s')
{-7} pm′‘rw_addr(a')
{-8} in_memory(min_linear, max_linear)(a')
{-9} Mem?(type_of(a'))
{-10} OK?(linear_resolve(a', Write)(s'))
{-11} OK?(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))
{-12} OK?(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s'))))))
{-13} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)(s')))))))
(state(linear_resolve(a', Read)
(state(memory_write(pm_phy‘mem)
(data(linear_resolve(a', Write)(s')), b')
(state(linear_resolve(a', Write)
(s'))))))))

{1} Mem?(data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a', Write)(s'))‘type_of)
∧
$0 \leq$
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a', Write)(s'))‘offset
∧
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a', Write)(s'))‘offset
$<$ max_linear_offset
{2} pm_phy‘rw_addr(data(linear_resolve(a', Write)(s')))

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

which is trivially true.

This completes the proof of `linear_plain_changed_memory_invariant.1.1.1.2.2`.

`linear_plain_changed_memory_invariant.1.1.2:`

```
{-1}  is_linear_plain_memory?(pm′) ⊃ pm′`states = pm_phy`states
{-2}  is_linear_plain_memory?(pm′) ⊃ plain_memory?(pm_phy)
{-3}  union(pm′`ro_addr, pm′`rw_addr)(a′) ∧
        is_linear_plain_memory?(pm′) ∧
          pm′`states(s′) ∧ OK?(linear_resolve(a′, Write)(s′))
        ⊃ pm_phy`states(state(linear_resolve(a′, Write)(s′)))
{-4}  b′ < max_byte
{-5}  is_linear_plain_memory?(pm′)
{-6}  pm′`states(s′)
{-7}  pm′`rw_addr(a′)
{-8}  in_memory(min_linear, max_linear)(a′)
{-9}  Mem?(type_of(a′))
{-10} OK?(linear_resolve(a′, Write)(s′))
{-11} OK?(memory_write(pm_phy`mem)
                (data(linear_resolve(a′, Write)(s′)), b′)
                (state(linear_resolve(a′, Write)(s′))))
{-12} OK?(linear_resolve(a′, Read)
                        (state(memory_write(pm_phy`mem)
                                           (data(linear_resolve(a′, Write)(s′)), b′)
                                           (state(linear_resolve(a′, Write)(s′))))))
{-13} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)(s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′))))))))
-------------------------------------------------------------------------------------------
{1}   union(pm′`ro_addr, pm′`rw_addr)(a′)
{2}   data(memory_read(pm_phy`mem)
                (data(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′)))))))
                (state(linear_resolve(a′, Read)
                                    (state(memory_write(pm_phy`mem)
                                                       (data(linear_resolve(a′, Write)(s′)), b′)
                                                       (state(linear_resolve(a′, Write)
                                                                            (s′))))))))
      = b′
```

Rewriting using union_right, matching in *,

This completes the proof of `linear_plain_changed_memory_invariant.1.1.2`.

`linear_plain_changed_memory_invariant.1.2:`

{-1}	$b' < \text{max\_byte}$
{-2}	is_linear_plain_memory?(pm′)
{-3}	pm′‘states(s')
{-4}	pm′‘rw_addr(a')
{-5}	in_memory(min_linear, max_linear)(a')
{-6}	OK?(memory_write(pm_phy‘mem)(a', b')(s'))
{-7}	OK?(memory_read(pm_phy‘mem)(a')(state(memory_write(pm_phy‘mem)(a', b')(s'))))
{1}	Mem?(type_of(a'))
{2}	data(memory_read(pm_phy‘mem)(a')(state(memory_write(pm_phy‘mem)(a', b')(s')))) $= b'$

Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using union_right, matching in *,
This completes the proof of `linear_plain_changed_memory_invariant.1.2`.
`linear_plain_changed_memory_invariant.2:`

{-1}	is_linear_plain_memory?(pm′)
{1}	transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))

Using lemma linear_plain_transformer_invariant,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets memory_write_transformers(pm!1‘mem, pm!1‘rw_addr), transformers_2 gets union(union(pm!1‘other_actions, union(memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr))), union(memory_read_side_effect_super_tr
union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_changed_memory_invariant.2`.
`linear_plain_changed_memory_invariant.3:`

{-1}	is_linear_plain_memory?(pm′)
{1}	transformer_invariant?(pm′‘states, memory_read_transformers(pm′‘mem, pm′‘rw_addr))

Using lemma linear_plain_transformer_invariant,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets memory_read_transformers(pm!1‘mem, pm!1‘rw_addr), transformers_2 gets union(union(pm!1‘other_actions, union(memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr))), union(memory_read_side_effect_super_tr
union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_changed_memory_invariant.3`.
Q.E.D.

C.116.7 Linear_Memory_Blessing_Properties.linear_plain_read_side_effect_unchanged

Terse proof for `linear_plain_read_side_effect_unchanged`.

`linear_plain_read_side_effect_unchanged:`

```
{1}  ∀ (pm: Plain_Memory[Physical_memory]):
       is_linear_plain_memory?(pm) ⊃
        side_effect_content_unchanged((pm`ro_addr ∪ pm`rw_addr), pm`states,
                                      memory_read_side_effect(pm`mem))
```

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Hiding formulas: -1,

Rewriting using pm_read_side_effect_linear, matching in *,

Expanding the definition of linear_read_side_effect,

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.1:`

```
{-1}  Mem?(type_of(a′))
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′`states(s′)
{-4}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-5}  OK?(IF null?(bl′) ∨
            (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
               a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
          THEN IF Mem?(type_of(a′))
                 THEN apply_side_effects(split(min_page, a′, bl′),
                                         linear_read_side_effect_in_page)
                                        (s′)
                 ELSE memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′)
                 ENDIF
          ELSE Fatal
          ENDIF)
-------------------------------------------------------------------------
{1}  IF null?(bl′) ∨
       (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
          a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
     THEN IF Mem?(type_of(a′))
            THEN data(apply_side_effects(split(min_page, a′, bl′),
                                         linear_read_side_effect_in_page)
                                        (s′))
            ELSE data(memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′))
            ENDIF
     ELSE data(Fatal)
     ENDIF
      = bl′
```

Case splitting on null?(bl!1),

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.1.1:`

```
{-1}  null?(bl′)
{-2}  Mem?(type_of(a′))
{-3}  is_linear_plain_memory?(pm′)
{-4}  pm′‘states(s′)
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6}  OK?(IF null?(bl′) ∨
            (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
               a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
          THEN IF Mem?(type_of(a′))
                 THEN apply_side_effects(split(min_page, a′, bl′),
                                         linear_read_side_effect_in_page)
                                        (s′)
                 ELSE memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′)
                 ENDIF
          ELSE Fatal
          ENDIF)
-------------------------------------------------------------------------
{1}   IF null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
      THEN IF Mem?(type_of(a′))
             THEN data(apply_side_effects(split(min_page, a′, bl′),
                                          linear_read_side_effect_in_page)
                                         (s′))
             ELSE data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′))
             ENDIF
      ELSE data(Fatal)
      ENDIF
       = bl′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of ok_result,

Expanding the definition of reduce,

Using lemma split_null,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.1`.

`linear_plain_read_side_effect_unchanged.1.2:`

```
{-1}  Mem?(type_of(a'))
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states(s')
{-4}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-5}  OK?(IF null?(bl') ∨
            (reg_base(min_linear)(type_of(a')) ≤ a' ∧
               a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
          THEN IF Mem?(type_of(a'))
                 THEN apply_side_effects(split(min_page, a', bl'),
                                         linear_read_side_effect_in_page)
                                        (s')
                 ELSE memory_read_side_effect(pm_phy`mem)(a', bl', cp')(s')
                 ENDIF
          ELSE Fatal
          ENDIF)
-------------------------------------------------------------------------
{1}   null?(bl')
{2}   IF null?(bl') ∨
         (reg_base(min_linear)(type_of(a')) ≤ a' ∧
            a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
       THEN IF Mem?(type_of(a'))
              THEN data(apply_side_effects(split(min_page, a', bl'),
                                           linear_read_side_effect_in_page)
                                          (s'))
              ELSE data(memory_read_side_effect(pm_phy`mem)(a', bl', cp')(s'))
              ENDIF
       ELSE data(Fatal)
       ENDIF
        = bl'
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma apply_side_effects_same_result,

Case splitting on offset(a!1) < max_linear_offset,

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.1.2.1:`

```
{-1}  offset(a′) < max_linear_offset
{-2}  is_linear_plain_memory?(pm′) ∧
       pm′‘states(s′) ∧
        (∀ (s1: (pm′‘states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]):
                       OK?(linear_read_side_effect_in_page(e)(s1)))
                   (split(min_page, a′, bl′)))
         ∧
         (∀ (s1: (pm′‘states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                        OK?(linear_read_side_effect_in_page(e)(s1)) ⊃
                          data(linear_read_side_effect_in_page(e)(s1)) = e‘2)
                    (split(min_page, a′, bl′)))
          ∧
          every(λ (e: [Memory_Address_4G, list[Byte]]):
                    transformer_invariant?(pm′‘states,
                                           singleton(expr_2_super(linear_read_side_effect_in_page
                                                                  (e)))))
                (split(min_page, a′, bl′))
       ⊃
       data(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
        =
        reduce(null,
               λ (e: [Address, list[Byte]], tail: list[Byte]):
                 LET head = e‘2 IN append(head, tail))
              (split(min_page, a′, bl′))
{-3}  Mem?(type_of(a′))
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7}  reg_base(min_linear)(type_of(a′)) ≤ a′
{-8}  a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-9}  OK?(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
------------------------------------------------------------------------------------------
{1}   null?(bl′)
{2}   data(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
       = bl′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

`linear_plain_read_side_effect_unchanged.1.2.1.1:`

{-1} offset(a') < max_linear_offset
{-2} is_linear_plain_memory?(pm$'$)
{-3} pm$'$`states(s')
{-4} data(apply_side_effects(split(min_page, a', bl$'$), linear_read_side_effect_in_page)(s'))
= reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
(split(min_page, a', bl$'$))
{-5} Mem?(type_of(a'))
{-6} (address_block(a', length(bl$'$)) ⊆ (pm$'$`ro_addr ∪ pm$'$`rw_addr))
{-7} reg_base(min_linear)(type_of(a')) ≤ a'
{-8} a' + length(bl$'$) ≤ reg_size(max_linear)(type_of(a'))
{-9} OK?(apply_side_effects(split(min_page, a', bl$'$), linear_read_side_effect_in_page)(s'))

{1} null?(bl$'$)
{2} data(apply_side_effects(split(min_page, a', bl$'$), linear_read_side_effect_in_page)(s'))
= bl$'$

Replacing using formula -4,

Using lemma split_pair_concat,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.1`.

`linear_plain_read_side_effect_unchanged.1.2.1.2:`

{-1} offset(a') < max_linear_offset
{-2} is_linear_plain_memory?(pm$'$)
{-3} pm$'$`states(s')
{-4} Mem?(type_of(a'))
{-5} (address_block(a', length(bl$'$)) ⊆ (pm$'$`ro_addr ∪ pm$'$`rw_addr))
{-6} reg_base(min_linear)(type_of(a')) ≤ a'
{-7} a' + length(bl$'$) ≤ reg_size(max_linear)(type_of(a'))
{-8} OK?(apply_side_effects(split(min_page, a', bl$'$), linear_read_side_effect_in_page)(s'))

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm$'$`states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
(split(min_page, a', bl$'$))
{2} null?(bl$'$)
{3} data(apply_side_effects(split(min_page, a', bl$'$), linear_read_side_effect_in_page)(s'))
= bl$'$

Rewriting using split_linear_read_side_effects_states, matching in *,

Keeping (-4 -6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.2`.

`linear_plain_read_side_effect_unchanged.1.2.1.3:`

{-1}	$\text{offset}(a') < \text{max\_linear\_offset}$
{-2}	$\text{is\_linear\_plain\_memory?}(\text{pm}')$
{-3}	$\text{pm}'\text{`states}(s')$
{-4}	$\text{Mem?}(\text{type\_of}(a'))$
{-5}	$(\text{address\_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{`ro\_addr} \cup \text{pm}'\text{`rw\_addr}))$
{-6}	$\text{reg\_base}(\text{min\_linear})(\text{type\_of}(a')) \leq a'$
{-7}	$a' + \text{length}(\text{bl}') \leq \text{reg\_size}(\text{max\_linear})(\text{type\_of}(a'))$
{-8}	$\text{OK?}(\text{apply\_side\_effects}(\text{split}(\text{min\_page}, a', \text{bl}'), \text{linear\_read\_side\_effect\_in\_page})(s'))$
{1}	$\forall\ (s_1 : (\text{pm}'\text{`states}))$: every($\lambda\ (e : [\text{Memory\_Address\_4G}, \text{list}[\text{Byte}]])$: OK?(linear_read_side_effect_in_page($e$)($s_1$)) $\supset$ data(linear_read_side_effect_in_page($e$)($s_1$)) $=$ $e$`2) (split(min_page, a', bl$'$))
{2}	$\text{null?}(\text{bl}')$
{3}	$\text{data}(\text{apply\_side\_effects}(\text{split}(\text{min\_page}, a', \text{bl}'), \text{linear\_read\_side\_effect\_in\_page})(s')) = \text{bl}'$

Repeatedly Skolemizing and flattening,

Using lemma split_linear_read_side_effects_data,

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.1.2.1.3.1:`

{-1}	$\text{is_linear_plain_memory?}(\text{pm}') \wedge (\text{address_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{`ro\_addr} \cup \text{pm}'\text{`rw_addr})) \wedge \text{pm}'\text{`states}(s_1') \supset$ every($\lambda\ (e : [\text{Memory\_Address\_4G}, \text{list}[\text{Byte}]])$: OK?(linear_read_side_effect_in_page($e$)($s_1'$)) $\supset$ data(linear_read_side_effect_in_page($e$)($s_1'$)) $=$ $e$`2) (split(min_page, a', bl$'$))
{-2}	$\text{pm}'\text{`states}(s_1')$
{-3}	$\text{offset}(a') < \text{max\_linear\_offset}$
{-4}	$\text{is\_linear\_plain\_memory?}(\text{pm}')$
{-5}	$\text{pm}'\text{`states}(s')$
{-6}	$\text{Mem?}(\text{type\_of}(a'))$
{-7}	$(\text{address\_block}(a', \text{length}(\text{bl}')) \subseteq (\text{pm}'\text{`ro\_addr} \cup \text{pm}'\text{`rw\_addr}))$
{-8}	$\text{reg\_base}(\text{min\_linear})(\text{type\_of}(a')) \leq a'$
{-9}	$a' + \text{length}(\text{bl}') \leq \text{reg\_size}(\text{max\_linear})(\text{type\_of}(a'))$
{-10}	$\text{OK?}(\text{apply\_side\_effects}(\text{split}(\text{min\_page}, a', \text{bl}'), \text{linear\_read\_side\_effect\_in\_page})(s'))$
{1}	every($\lambda\ (e : [\text{Memory\_Address\_4G}, \text{list}[\text{Byte}]])$: OK?(linear_read_side_effect_in_page(e)(s_1')) $\supset$ data(linear_read_side_effect_in_page(e)(s_1')) $=$ e`2) (split(min_page, a', bl$'$))
{2}	$\text{null?}(\text{bl}')$
{3}	$\text{data}(\text{apply\_side\_effects}(\text{split}(\text{min\_page}, a', \text{bl}'), \text{linear\_read\_side\_effect\_in\_page})(s')) = \text{bl}'$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.3.1`.

`linear_plain_read_side_effect_unchanged.1.2.1.3.2:`

{-1} pm′‘states(s'_1)
{-2} offset(a') < max_linear_offset
{-3} is_linear_plain_memory?(pm′)
{-4} pm′‘states(s')
{-5} Mem?(type_of(a'))
{-6} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7} reg_base(min_linear)(type_of(a')) ≤ a'
{-8} a' + length(bl′) ≤ reg_size(max_linear)(type_of(a'))
{-9} OK?(apply_side_effects(split(min_page, a', bl′), linear_read_side_effect_in_page)(s'))

{1} 0 ≤ a'‘offset
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s'_1)) ⊃
data(linear_read_side_effect_in_page(e)(s'_1)) = e‘2)
(split(min_page, a', bl′))
{3} null?(bl′)
{4} data(apply_side_effects(split(min_page, a', bl′), linear_read_side_effect_in_page)(s'))
= bl′

Keeping (-5 -7 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.3.2`.

`linear_plain_read_side_effect_unchanged.1.2.1.4:`

{-1} offset(a') < max_linear_offset
{-2} is_linear_plain_memory?(pm′)
{-3} pm′‘states(s')
{-4} Mem?(type_of(a'))
{-5} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6} reg_base(min_linear)(type_of(a')) ≤ a'
{-7} a' + length(bl′) ≤ reg_size(max_linear)(type_of(a'))
{-8} OK?(apply_side_effects(split(min_page, a', bl′), linear_read_side_effect_in_page)(s'))

{1} ∀ (s_1: (pm′‘states)):
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s_1)))
(split(min_page, a', bl′))
{2} null?(bl′)
{3} data(apply_side_effects(split(min_page, a', bl′), linear_read_side_effect_in_page)(s'))
= bl′

Repeatedly Skolemizing and flattening,

Using lemma split_linear_read_side_effects_ok,

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.1.2.1.4.1:`

```
{-1}  is_linear_plain_memory?(pm′) ∧
       (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr)) ∧
        pm′‘states(s′1)
       ⊃
       every(λ (e: [Memory_Address_4G, list[Byte]]):
                 OK?(linear_read_side_effect_in_page(e)(s′1)))
             (split(min_page, a′, bl′))
{-2}  pm′‘states(s′1)
{-3}  offset(a′) < max_linear_offset
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  Mem?(type_of(a′))
{-7}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8}  reg_base(min_linear)(type_of(a′)) ≤ a′
{-9}  a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-10} OK?(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
-------------------------------------------------------------------------------
{1}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                 OK?(linear_read_side_effect_in_page(e)(s′1)))
             (split(min_page, a′, bl′))
{2}   null?(bl′)
{3}   data(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
       = bl′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.4.1`.

`linear_plain_read_side_effect_unchanged.1.2.1.4.2:`

```
{-1}  pm′‘states(s′1)
{-2}  offset(a′) < max_linear_offset
{-3}  is_linear_plain_memory?(pm′)
{-4}  pm′‘states(s′)
{-5}  Mem?(type_of(a′))
{-6}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7}  reg_base(min_linear)(type_of(a′)) ≤ a′
{-8}  a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-9}  OK?(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
-------------------------------------------------------------------------------
{1}   0 ≤ a′‘offset
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                 OK?(linear_read_side_effect_in_page(e)(s′1)))
             (split(min_page, a′, bl′))
{3}   null?(bl′)
{4}   data(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
       = bl′
```

Keeping (-5 -7 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.1.4.2`.

`linear_plain_read_side_effect_unchanged.1.2.2:`

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
(∀ (s1: (pm′‘states)):
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s1)))
(split(min_page, a′, bl′)))
∧
(∀ (s1: (pm′‘states)):
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s1)) ⊃
data(linear_read_side_effect_in_page(e)(s1)) = e‘2)
(split(min_page, a′, bl′)))
∧
every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page
(e)))))
(split(min_page, a′, bl′))
⊃
data(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
=
reduce(null,
λ (e: [Address, list[Byte]], tail: list[Byte]):
LET head = e‘2 IN append(head, tail))
(split(min_page, a′, bl′))
{-2} Mem?(type_of(a′))
{-3} is_linear_plain_memory?(pm′)
{-4} pm′‘states(s′)
{-5} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6} reg_base(min_linear)(type_of(a′)) ≤ a′
{-7} a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-8} OK?(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))

{1} offset(a′) < max_linear_offset
{2} null?(bl′)
{3} data(apply_side_effects(split(min_page, a′, bl′), linear_read_side_effect_in_page)(s′))
= bl′

Keeping (-2 -5 -6 -7 1 2) and hiding *,

Expanding the definition of length,

Installing automatic rewrites from: Mem max_linear reg_size

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.1.2.2`.

`linear_plain_read_side_effect_unchanged.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states(s′)
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4}  OK?(IF null?(bl′) ∨
            (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
               a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
          THEN IF Mem?(type_of(a′))
                 THEN apply_side_effects(split(min_page, a′, bl′),
                                         linear_read_side_effect_in_page)
                                        (s′)
                 ELSE memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′)
                 ENDIF
          ELSE Fatal
          ENDIF)
-------------------------------------------------------------------------
{1}   Mem?(type_of(a′))
{2}   IF null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
      THEN IF Mem?(type_of(a′))
             THEN data(apply_side_effects(split(min_page, a′, bl′),
                                          linear_read_side_effect_in_page)
                                         (s′))
             ELSE data(memory_read_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′))
             ENDIF
      ELSE data(Fatal)
      ENDIF
       = bl′
```

Using lemma pm_memory_addr,

Case splitting on null?(bl!1),

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.2.1:`

```
{-1}  null?(bl′)
{-2}  is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
       in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-3}  is_linear_plain_memory?(pm′)
{-4}  pm′`states(s′)
{-5}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-6}  OK?(IF null?(bl′) ∨
              (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
                  a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
            THEN IF Mem?(type_of(a′))
                   THEN apply_side_effects(split(min_page, a′, bl′),
                                                 linear_read_side_effect_in_page)
                                           (s′)
                   ELSE memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′)
                   ENDIF
            ELSE Fatal
            ENDIF)
-------------------------------------------------------------------------------
{1}   Mem?(type_of(a′))
{2}   IF null?(bl′) ∨
         (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
            a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
       THEN IF Mem?(type_of(a′))
              THEN data(apply_side_effects(split(min_page, a′, bl′),
                                             linear_read_side_effect_in_page)
                                           (s′))
              ELSE data(memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′))
              ENDIF
       ELSE data(Fatal)
       ENDIF
        = bl′
```

Simplifying, rewriting, and recording with decision procedures,

Using lemma pm_plain_phy,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_side_effect_content_unchanged_read_block,

Expanding the definition of side_effect_content_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_read_side_effect_unchanged.2.1.1:`

{-1}	∀ (s: Physical_memory, a: Address, bl: list[Byte], cp: bool): pm_phy‘states(s) ∧ (address_block(a, length(bl)) ⊆ address_block(a', length(bl′))) ∧ OK?(memory_read_side_effect(pm_phy‘mem)(a, bl, cp)(s)) ⊃ data(memory_read_side_effect(pm_phy‘mem)(a, bl, cp)(s)) = bl
{-2}	is_linear_plain_memory?(pm′)
{-3}	pm′‘states = pm_phy‘states
{-4}	plain_memory?(pm_phy)
{-5}	null?(bl′)
{-6}	pm′‘states(s')
{-7}	(address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8}	OK?(memory_read_side_effect(pm_phy‘mem)(a', bl′, cp′)(s'))
{1}	union(pm′‘ro_addr, pm′‘rw_addr)(a')
{2}	Mem?(type_of(a'))
{3}	data(memory_read_side_effect(pm_phy‘mem)(a', bl′, cp′)(s')) = bl′

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.2.1.1`.

`linear_plain_read_side_effect_unchanged.2.1.2:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	pm′‘states = pm_phy‘states
{-3}	plain_memory?(pm_phy)
{-4}	null?(bl′)
{-5}	pm′‘states(s')
{-6}	(address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7}	OK?(memory_read_side_effect(pm_phy‘mem)(a', bl′, cp′)(s'))
{1}	(address_block(a', length(bl′)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}	union(pm′‘ro_addr, pm′‘rw_addr)(a')
{3}	Mem?(type_of(a'))
{4}	data(memory_read_side_effect(pm_phy‘mem)(a', bl′, cp′)(s')) = bl′

Hiding formulas: (-1 -2 -3 -7 4),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_read_side_effect_unchanged.2.1.2`.

`linear_plain_read_side_effect_unchanged.2.2:`

{-1} is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-2} is_linear_plain_memory?(pm′)
{-3} pm′`states(s′)
{-4} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-5} OK?(IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN apply_side_effects(split(min_page, a′, bl′),
linear_read_side_effect_in_page)
(s′)
ELSE memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′)
ENDIF
ELSE Fatal
ENDIF)

{1} null?(bl′)
{2} Mem?(type_of(a′))
{3} IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN data(apply_side_effects(split(min_page, a′, bl′),
linear_read_side_effect_in_page)
(s′))
ELSE data(memory_read_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′))
ENDIF
ELSE data(Fatal)
ENDIF
= bl′

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1 2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_read_side_effect_unchanged.2.2`.
Q.E.D.

C.116.8 Linear_Memory_Blessing_Properties.linear_plain_write_side_effect_unchanged

Terse proof for `linear_plain_write_side_effect_unchanged`.

`linear_plain_write_side_effect_unchanged:`

{1} ∀ (pm: Plain_Memory[Physical_memory]):
is_linear_plain_memory?(pm) ⊃
side_effect_content_unchanged(pm`rw_addr, pm`states, memory_write_side_effect(pm`mem))

Expanding the definition of side_effect_content_unchanged,
Repeatedly Skolemizing and flattening,
Hiding formulas: -1,

Rewriting using pm_write_side_effect_linear, matching in *,

Expanding the definition of linear_write_side_effect,

Case splitting on Mem?(type_of(a!1)),

we get 2 subgoals:

`linear_plain_write_side_effect_unchanged.1:`

```
{-1}  Mem?(type_of(a'))
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states(s')
{-4}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-5}  OK?(IF null?(bl') ∨
              (reg_base(min_linear)(type_of(a')) ≤ a' ∧
                  a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
            THEN IF Mem?(type_of(a'))
                   THEN apply_side_effects(split(min_page, a', bl'),
                                                 linear_write_side_effect_in_page)
                                           (s')
                   ELSE memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s')
                   ENDIF
            ELSE Fatal
            ENDIF)
-------------------------------------------------------------------------------
{1}   IF null?(bl') ∨
         (reg_base(min_linear)(type_of(a')) ≤ a' ∧
            a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
       THEN IF Mem?(type_of(a'))
              THEN data(apply_side_effects(split(min_page, a', bl'),
                                                 linear_write_side_effect_in_page)
                                           (s'))
              ELSE data(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'))
              ENDIF
       ELSE data(Fatal)
       ENDIF
        = bl'
```

Case splitting on null?(bl!1),

we get 2 subgoals:

`linear_plain_write_side_effect_unchanged.1.1:`

```
{-1}  null?(bl′)
{-2}  Mem?(type_of(a′))
{-3}  is_linear_plain_memory?(pm′)
{-4}  pm′`states(s′)
{-5}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-6}  OK?(IF null?(bl′) ∨
            (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
               a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
          THEN IF Mem?(type_of(a′))
                 THEN apply_side_effects(split(min_page, a′, bl′),
                                         linear_write_side_effect_in_page)
                                        (s′)
                 ELSE memory_write_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′)
                 ENDIF
          ELSE Fatal
          ENDIF)
-------------------------------------------------------------------------------
{1}   IF null?(bl′) ∨
        (reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
           a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
      THEN IF Mem?(type_of(a′))
             THEN data(apply_side_effects(split(min_page, a′, bl′),
                                          linear_write_side_effect_in_page)
                                         (s′))
             ELSE data(memory_write_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′))
             ENDIF
      ELSE data(Fatal)
      ENDIF
       = bl′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of ok_result,

Expanding the definition of reduce,

Using lemma split_null,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.1`.

`linear_plain_write_side_effect_unchanged.1.2:`

```
{-1}  Mem?(type_of(a'))
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states(s')
{-4}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-5}  OK?(IF null?(bl') ∨
            (reg_base(min_linear)(type_of(a')) ≤ a' ∧
               a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
          THEN IF Mem?(type_of(a'))
                 THEN apply_side_effects(split(min_page, a', bl'),
                                         linear_write_side_effect_in_page)
                                        (s')
                 ELSE memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s')
                 ENDIF
          ELSE Fatal
          ENDIF)
-------------------------------------------------------------------------
{1}   null?(bl')
{2}   IF null?(bl') ∨
         (reg_base(min_linear)(type_of(a')) ≤ a' ∧
            a' + length(bl') ≤ reg_size(max_linear)(type_of(a')))
       THEN IF Mem?(type_of(a'))
              THEN data(apply_side_effects(split(min_page, a', bl'),
                                           linear_write_side_effect_in_page)
                                          (s'))
              ELSE data(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'))
              ENDIF
       ELSE data(Fatal)
       ENDIF
        = bl'
```

Expanding the definition of reg_base,

Expanding the definition of <=,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Using lemma apply_side_effects_same_result,

Case splitting on offset(a!1) < max_linear_offset,

we get 2 subgoals:

`linear_plain_write_side_effect_unchanged.1.2.1:`

```
{-1}  offset(a′) < max_linear_offset
{-2}  is_linear_plain_memory?(pm′) ∧
        pm′‘states(s′) ∧
         (∀ (s1: (pm′‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                      OK?(linear_write_side_effect_in_page(e)(s1)))
                  (split(min_page, a′, bl′)))
          ∧
          (∀ (s1: (pm′‘states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]):
                       OK?(linear_write_side_effect_in_page(e)(s1)) ⊃
                         data(linear_write_side_effect_in_page(e)(s1)) = e‘2)
                   (split(min_page, a′, bl′)))
           ∧
           every(λ (e: [Memory_Address_4G, list[Byte]]):
                     transformer_invariant?(pm′‘states,
                                            singleton(expr_2_super(linear_write_side_effect_in_page
                                                                   (e)))))
                 (split(min_page, a′, bl′))
        ⊃
        data(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)
                               (s′))
         =
         reduce(null,
                λ (e: [Address, list[Byte]], tail: list[Byte]):
                  LET head = e‘2 IN append(head, tail))
               (split(min_page, a′, bl′))
{-3}  Mem?(type_of(a′))
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-7}  Mem_ = a′‘type_of
{-8}  0 ≤ a′‘offset
{-9}  (a′ + length(bl′))‘type_of = reg_size(max_linear)(type_of(a′))‘type_of
{-10} (a′ + length(bl′))‘offset ≤ reg_size(max_linear)(type_of(a′))‘offset
{-11} OK?(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
-------------------------------------------------------------------------------
 {1}  null?(bl′)
 {2}  data(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
       = bl′
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

`linear_plain_write_side_effect_unchanged.1.2.1.1:`

{-1}	offset(a') $<$ max_linear_offset
{-2}	is_linear_plain_memory?(pm$'$)
{-3}	pm$'$`states(s')
{-4}	data(apply_side_effects(split(min_page, a', bl$'$), linear_write_side_effect_in_page)(s')) = reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail)) (split(min_page, a', bl$'$))
{-5}	Mem?(type_of(a'))
{-6}	(address_block(a', length(bl$'$)) $\subseteq$ pm$'$`rw_addr)
{-7}	$0 \leq a'$`offset
{-8}	(a' + length(bl$'$))`type_of = reg_size(max_linear)(type_of($a'$))`type_of
{-9}	(a' + length(bl$'$))`offset $\leq$ reg_size(max_linear)(type_of($a'$))`offset
{-10}	OK?(apply_side_effects(split(min_page, a', bl$'$), linear_write_side_effect_in_page)(s'))
{1}	null?(bl$'$)
{2}	data(apply_side_effects(split(min_page, a', bl$'$), linear_write_side_effect_in_page)(s')) = bl$'$

Replacing using formula -4,

Using lemma split_pair_concat,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.1.1`.

`linear_plain_write_side_effect_unchanged.1.2.1.2:`

{-1}	offset(a') $<$ max_linear_offset
{-2}	is_linear_plain_memory?(pm$'$)
{-3}	pm$'$`states(s')
{-4}	Mem?(type_of(a'))
{-5}	(address_block(a', length(bl$'$)) $\subseteq$ pm$'$`rw_addr)
{-6}	$0 \leq a'$`offset
{-7}	(a' + length(bl$'$))`type_of = reg_size(max_linear)(type_of($a'$))`type_of
{-8}	(a' + length(bl$'$))`offset $\leq$ reg_size(max_linear)(type_of($a'$))`offset
{-9}	OK?(apply_side_effects(split(min_page, a', bl$'$), linear_write_side_effect_in_page)(s'))
{1}	every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm$'$`states, singleton(expr_2_super(linear_write_side_effect_in_page(e)))) (split(min_page, a', bl$'$))
{2}	null?(bl$'$)
{3}	data(apply_side_effects(split(min_page, a', bl$'$), linear_write_side_effect_in_page)(s')) = bl$'$

Rewriting using split_linear_write_side_effects_states, matching in *,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.1.2`.

`linear_plain_write_side_effect_unchanged.1.2.1.3:`

```
{-1}  offset(a′) < max_linear_offset
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  Mem?(type_of(a′))
{-5}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-6}  0 ≤ a′‘offset
{-7}  (a′ + length(bl′))‘type_of = reg_size(max_linear)(type_of(a′))‘type_of
{-8}  (a′ + length(bl′))‘offset ≤ reg_size(max_linear)(type_of(a′))‘offset
{-9}  OK?(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
------------------------------------------------------------------------------
{1}   ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s₁)) ⊃
                  data(linear_write_side_effect_in_page(e)(s₁)) = e‘2)
              (split(min_page, a′, bl′))
{2}   null?(bl′)
{3}   data(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
        = bl′
```

Repeatedly Skolemizing and flattening,

Using lemma split_linear_write_side_effects_data,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.1.3`.

`linear_plain_write_side_effect_unchanged.1.2.1.4:`

```
{-1}  offset(a′) < max_linear_offset
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  Mem?(type_of(a′))
{-5}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-6}  0 ≤ a′‘offset
{-7}  (a′ + length(bl′))‘type_of = reg_size(max_linear)(type_of(a′))‘type_of
{-8}  (a′ + length(bl′))‘offset ≤ reg_size(max_linear)(type_of(a′))‘offset
{-9}  OK?(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
------------------------------------------------------------------------------
{1}   ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s₁)))
              (split(min_page, a′, bl′))
{2}   null?(bl′)
{3}   data(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
        = bl′
```

Repeatedly Skolemizing and flattening,

Using lemma split_linear_write_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.1.4`.

`linear_plain_write_side_effect_unchanged.1.2.2:`

```
{-1}   is_linear_plain_memory?(pm′) ∧
        pm′‘states(s′) ∧
         (∀ (s1: (pm′‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                     OK?(linear_write_side_effect_in_page(e)(s1)))
                  (split(min_page, a′, bl′)))
          ∧
          (∀ (s1: (pm′‘states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]):
                      OK?(linear_write_side_effect_in_page(e)(s1)) ⊃
                       data(linear_write_side_effect_in_page(e)(s1)) = e‘2)
                   (split(min_page, a′, bl′)))
           ∧
           every(λ (e: [Memory_Address_4G, list[Byte]]):
                    transformer_invariant?(pm′‘states,
                                           singleton(expr_2_super(linear_write_side_effect_in_pag
                                                                  (e)))))
                 (split(min_page, a′, bl′))
        ⊃
        data(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)
                               (s′))
         =
         reduce(null,
                λ (e: [Address, list[Byte]], tail: list[Byte]):
                  LET head = e‘2 IN append(head, tail))
               (split(min_page, a′, bl′))
{-2}   Mem?(type_of(a′))
{-3}   is_linear_plain_memory?(pm′)
{-4}   pm′‘states(s′)
{-5}   (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-6}   Mem_ = a′‘type_of
{-7}   0 ≤ a′‘offset
{-8}   (a′ + length(bl′))‘type_of = reg_size(max_linear)(type_of(a′))‘type_of
{-9}   (a′ + length(bl′))‘offset ≤ reg_size(max_linear)(type_of(a′))‘offset
{-10}  OK?(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
-------------------------------------------------------------------------------
 {1}   offset(a′) < max_linear_offset
 {2}   null?(bl′)
 {3}   data(apply_side_effects(split(min_page, a′, bl′), linear_write_side_effect_in_page)(s′))
        = bl′
```

Keeping (-6 -9 1 2) and hiding *,

Expanding the definition of length,

Installing automatic rewrites from: Mem max_linear

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.1.2.2`.

`linear_plain_write_side_effect_unchanged.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-4} OK?(IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN apply_side_effects(split(min_page, a′, bl′),
linear_write_side_effect_in_page)
(s′)
ELSE memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′)
ENDIF
ELSE Fatal
ENDIF)

{1} Mem?(type_of(a′))
{2} IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN data(apply_side_effects(split(min_page, a′, bl′),
linear_write_side_effect_in_page)
(s′))
ELSE data(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′))
ENDIF
ELSE data(Fatal)
ENDIF
= bl′

Using lemma pm_memory_addr,

Case splitting on null?(bl!1),

we get 2 subgoals:

`linear_plain_write_side_effect_unchanged.2.1:`

{-1} null?(bl′)
{-2} is_linear_plain_memory?(pm′) ∧ union(pm′`ro_addr, pm′`rw_addr)(a′) ⊃
in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-3} is_linear_plain_memory?(pm′)
{-4} pm′`states(s′)
{-5} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-6} OK?(IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN apply_side_effects(split(min_page, a′, bl′),
linear_write_side_effect_in_page)
(s′)
ELSE memory_write_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′)
ENDIF
ELSE Fatal
ENDIF)

{1} Mem?(type_of(a′))
{2} IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN data(apply_side_effects(split(min_page, a′, bl′),
linear_write_side_effect_in_page)
(s′))
ELSE data(memory_write_side_effect(pm_phy`mem)(a′, bl′, cp′)(s′))
ENDIF
ELSE data(Fatal)
ENDIF
= bl′

Simplifying, rewriting, and recording with decision procedures,

Using lemma pm_plain_phy,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_side_effect_content_unchanged_write_block,

Expanding the definition of side_effect_content_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_plain_write_side_effect_unchanged.2.1.1:`

{-1} $\forall$ (s: Physical_memory, a: Address, bl: list[Byte], cp: bool):
pm_phy`states(s) $\wedge$
(address_block(a, length(bl)) $\subseteq$ address_block(a', length(bl'))) $\wedge$
OK?(memory_write_side_effect(pm_phy`mem)(a, bl, cp)(s))
$\supset$ data(memory_write_side_effect(pm_phy`mem)(a, bl, cp)(s)) = bl
{-2} is_linear_plain_memory?(pm')
{-3} pm'`states = pm_phy`states
{-4} plain_memory?(pm_phy)
{-5} null?(bl')
{-6} pm'`states(s')
{-7} (address_block(a', length(bl')) $\subseteq$ pm'`rw_addr)
{-8} OK?(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'))

{1} union(pm'`ro_addr, pm'`rw_addr)(a')
{2} Mem?(type_of(a'))
{3} data(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s')) = bl'

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.2.1.1`.

`linear_plain_write_side_effect_unchanged.2.1.2:`

{-1} is_linear_plain_memory?(pm')
{-2} pm'`states = pm_phy`states
{-3} plain_memory?(pm_phy)
{-4} null?(bl')
{-5} pm'`states(s')
{-6} (address_block(a', length(bl')) $\subseteq$ pm'`rw_addr)
{-7} OK?(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s'))

{1} (address_block(a', length(bl')) $\subseteq$ pm_phy`rw_addr)
{2} union(pm'`ro_addr, pm'`rw_addr)(a')
{3} Mem?(type_of(a'))
{4} data(memory_write_side_effect(pm_phy`mem)(a', bl', cp')(s')) = bl'

Hiding formulas: (-1 -2 -3 -7 4),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_plain_write_side_effect_unchanged.2.1.2`.

`linear_plain_write_side_effect_unchanged.2.2:`

{-1} is_linear_plain_memory?(pm′) ∧ union(pm′‘ro_addr, pm′‘rw_addr)(a′) ⊃
in_memory(min_linear, max_linear)(a′) ∧ Mem?(type_of(a′))
{-2} is_linear_plain_memory?(pm′)
{-3} pm′‘states(s′)
{-4} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-5} OK?(IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN apply_side_effects(split(min_page, a′, bl′),
linear_write_side_effect_in_page)
(s′)
ELSE memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′)
ENDIF
ELSE Fatal
ENDIF)

{1} null?(bl′)
{2} Mem?(type_of(a′))
{3} IF null?(bl′) ∨
(reg_base(min_linear)(type_of(a′)) ≤ a′ ∧
a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′)))
THEN IF Mem?(type_of(a′))
THEN data(apply_side_effects(split(min_page, a′, bl′),
linear_write_side_effect_in_page)
(s′))
ELSE data(memory_write_side_effect(pm_phy‘mem)(a′, bl′, cp′)(s′))
ENDIF
ELSE data(Fatal)
ENDIF
= bl′

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1 2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_plain_write_side_effect_unchanged.2.2`.
Q.E.D.

C.116.9 Linear_Memory_Blessing_Properties.linear_memory_plain_memory

Terse proof for `linear_memory_plain_memory`.
`linear_memory_plain_memory:`

{1} ∀ (pm: Plain_Memory⌊Physical_memory⌋): is_linear_plain_memory?(pm) ⊃ plain_memory?(pm)

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: (linear_plain_transformers_ok! linear_plain_unchanged_memory_invariant! linear_plain_unchanged_memory_invariant_write! linear_plain_unchanged_memory_write_invariant! linear_plain_changed_memory_invariant! linear_plain_read_side_effect_unchanged! linear_plain_write_side_effect_uncha
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_memory_plain_memory`.
Q.E.D.

C.117 Proofs for Linear_Memory_Properties (challenge-linear.pvs)

C.117.1 Linear_Memory_Properties.pm_read_linear

Terse proof for `pm_read_linear`.
`pm_read_linear`:

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Address):
is_linear_plain_memory?(pm) ⊃ memory_read(pm`mem)(a) = linear_read(a)

Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Expanding the definition of linear_pm,
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_read_linear`.
Q.E.D.

C.117.2 Linear_Memory_Properties.pm_write_linear

Terse proof for `pm_write_linear`.
`pm_write_linear`:

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Address, b: Byte):
is_linear_plain_memory?(pm) ⊃ memory_write(pm`mem)(a, b) = linear_write(a, b)

Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Replacing using formula -2,
Expanding the definition of linear_pm,
which is trivially true.
This completes the proof of `pm_write_linear`.
Q.E.D.

C.117.3 Linear_Memory_Properties.pm_read_side_effect_linear

Terse proof for `pm_read_side_effect_linear`.
`pm_read_side_effect_linear`:

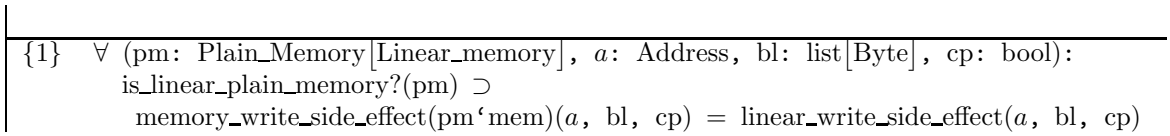
{1} ∀ (pm: Plain_Memory[Linear_memory], a: Address, bl: list[Byte], cp: bool):
is_linear_plain_memory?(pm) ⊃
memory_read_side_effect(pm`mem)(a, bl, cp) = linear_read_side_effect(a, bl, cp)

Repeatedly Skolemizing and flattening,

Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Replacing using formula -2,
Expanding the definition of linear_pm,
which is trivially true.
This completes the proof of `pm_read_side_effect_linear`.
Q.E.D.

C.117.4 Linear_Memory_Properties.pm_write_side_effect_linear

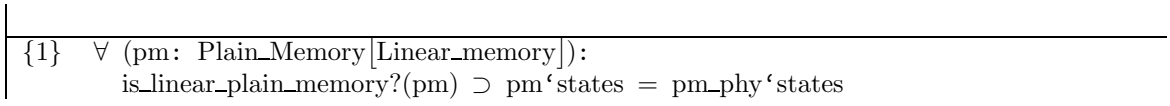
Terse proof for `pm_write_side_effect_linear`.
`pm_write_side_effect_linear:`



Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Expanding the definition of linear_pm,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_write_side_effect_linear`.
Q.E.D.

C.117.5 Linear_Memory_Properties.pm_states

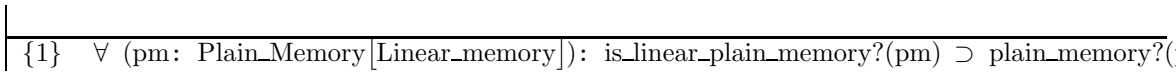
Terse proof for `pm_states`.
`pm_states:`



Expanding the definition of is_linear_plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of `pm_states`.
Q.E.D.

C.117.6 Linear_Memory_Properties.pm_plain_phy

Terse proof for `pm_plain_phy`.
`pm_plain_phy:`

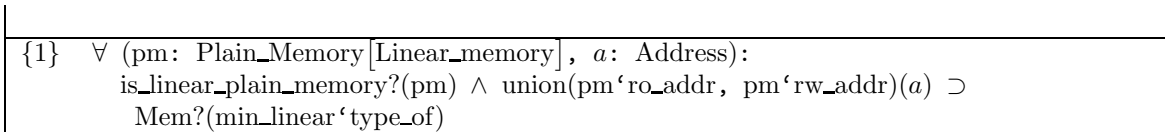


Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_plain_phy`.
Q.E.D.

C.117.7 Linear_Memory_Properties.pm_memory_addr_TCC1

Terse proof for `pm_memory_addr_TCC1`.

`pm_memory_addr_TCC1`:

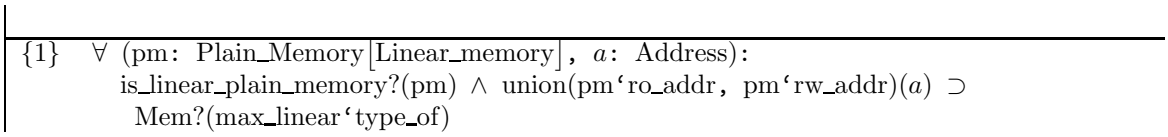


Repeatedly Skolemizing and flattening,
Expanding the definition of min_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of `pm_memory_addr_TCC1`.
Q.E.D.

C.117.8 Linear_Memory_Properties.pm_memory_addr_TCC2

Terse proof for `pm_memory_addr_TCC2`.

`pm_memory_addr_TCC2`:

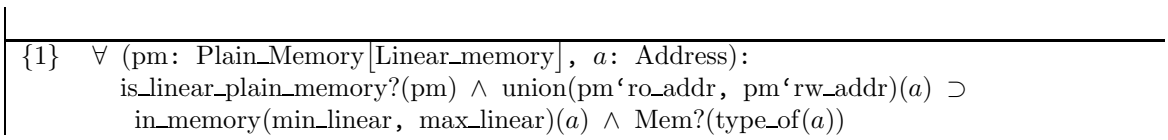


Repeatedly Skolemizing and flattening,
Expanding the definition of max_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of `pm_memory_addr_TCC2`.
Q.E.D.

C.117.9 Linear_Memory_Properties.pm_memory_addr

Terse proof for `pm_memory_addr`.

`pm_memory_addr`:



Expanding the definition of is_linear_plain_memory?,
Repeatedly Skolemizing and flattening,
Expanding the definition of max_linear,
Expanding the definition of Mem,
Instantiating quantified variables,
Keeping (-13 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_memory_addr`.
Q.E.D.

C.117.10 Linear_Memory_Properties.pm_linear_blessed

Terse proof for `pm_linear_blessed`.
`pm_linear_blessed:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy]):
       is_linear_plain_memory?(pm) ∧ pm`states(s) ⊃
         linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm`ro_addr),
                            restrict[Address, Memory_Address_4G, boolean](pm`rw_addr))
```

Expanding the definition of is_linear_plain_memory?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `pm_linear_blessed`.
Q.E.D.

C.117.11 Linear_Memory_Properties.pm_linear_resolve_read_ok_TCC1

Terse proof for `pm_linear_resolve_read_ok_TCC1`.
`pm_linear_resolve_read_ok_TCC1:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: (pm`states),
          a: ((pm`ro_addr ∪ pm`rw_addr))):
       is_linear_plain_memory?(pm) ⊃
         Mem?(a`type_of) ∧ 0 ≤ a`offset ∧ a`offset < max_linear_offset
```

Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `pm_linear_resolve_read_ok_TCC1`.
Q.E.D.

C.117.12 Linear_Memory_Properties.pm_linear_resolve_read_ok

Terse proof for `pm_linear_resolve_read_ok`.
`pm_linear_resolve_read_ok:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: (pm`states),
          a: ((pm`ro_addr ∪ pm`rw_addr))):
       is_linear_plain_memory?(pm) ⊃ OK?(linear_resolve(a, Read)(s))
```

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of linear_blessed?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-9 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_linear_resolve_read_ok`.
Q.E.D.

C.117.13 Linear_Memory_Properties.pm_linear_resolve_write_ok_TCC1

Terse proof for `pm_linear_resolve_write_ok_TCC1`.
`pm_linear_resolve_write_ok_TCC1`:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: (pm‘states), a: (pm‘rw_addr)):
is_linear_plain_memory?(pm) ⊃
Mem?(a‘type_of) ∧ 0 ≤ a‘offset ∧ a‘offset < max_linear_offset

Expanding the definition of is_linear_plain_memory?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Keeping (-1 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_linear_resolve_write_ok_TCC1`.
Q.E.D.

C.117.14 Linear_Memory_Properties.pm_linear_resolve_write_ok

Terse proof for `pm_linear_resolve_write_ok`.
`pm_linear_resolve_write_ok`:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: (pm‘states), a: (pm‘rw_addr)):
is_linear_plain_memory?(pm) ⊃ OK?(linear_resolve(a, Write)(s))

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of linear_blessed?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-9 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_linear_resolve_write_ok`.
Q.E.D.

C.117.15 Linear_Memory_Properties.pm_linear_resolve_reg_transformers_other

Terse proof for `pm_linear_resolve_reg_transformers_other`.
`pm_linear_resolve_reg_transformers_other`:

{1} ∀ (pm: Plain_Memory[Linear_memory]):
is_linear_plain_memory?(pm) ⊃
(linear_resolve_register_transformers ⊆ pm_phy‘other_actions)

Expanding the definition of is_linear_plain_memory?,

Repeatedly Skolemizing and flattening,

This completes the proof of `pm_linear_resolve_reg_transformers_other`.

Q.E.D.

C.117.16 Linear_Memory_Properties.pm_unchanged_singleton_linear_resolve_reg_transfor

Terse proof for `pm_unchanged_singleton_linear_resolve_reg_transformers`.

`pm_unchanged_singleton_linear_resolve_reg_transformers:`

$$\{1\}\quad \forall\ (\mathrm{pm}:\ \mathrm{Plain\_Memory}[\mathrm{Linear\_memory}],\ q:\ [\mathrm{Physical\_memory} \to \mathrm{SuperResult}[\mathrm{Physical\_memory}]]):\ \mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}) \wedge \mathrm{linear\_resolve\_register\_transformers}(q) \supset \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm\_phy`mem},\ \mathrm{pm\_phy`states},\ \mathrm{singleton}(q),\ (\mathrm{pm\_phy`ro\_addr} \cup \mathrm{pm\_phy`rw\_addr}))$$

Repeatedly Skolemizing and flattening,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

pm_unchanged_singleton_linear_resolve_reg_transformers.1:

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} pm′‘mem = linear_pm
{-3} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-5} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-6} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-7} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-8} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-9} ∀ (s1, s2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-10} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-11} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-12} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-13} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-14} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
pm_phy‘ro_addr)
{-15} unchanged_memory_write_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-16} changed_memory_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-17} transformers_ok?(pm_phy‘states,
((memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)) ∪ me
{-18} side_effect_content_unchanged((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), pm_phy‘states,
memory_read_side_effect(pm_phy‘mem))
{-19} side_effect_content_unchanged(pm_phy‘rw_addr, pm_phy‘states,
memory_write_side_effect(pm_phy‘mem))
{-20} pm′‘states = pm_phy‘states
{-21} linear_resolve_register_transformers(q′)

{1} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Rewriting using subset_reflexive, matching in *,

This completes the proof of `pm_unchanged_singleton_linear_resolve_reg_transformers.1`.

pm_unchanged_singleton_linear_resolve_reg_transformers.2:

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_p
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} pm′‘mem = linear_pm
{-3} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-5} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-6} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-7} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-8} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-9} ∀ (s1, s2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-10} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-11} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-12} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-13} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-14} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
memory_write_transformers(pm_phy‘mem, pm_phy‘rw_addr),
pm_phy‘ro_addr)
{-15} unchanged_memory_write_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-16} changed_memory_invariant?(pm_phy‘mem, pm_phy‘states, pm_phy‘rw_addr)
{-17} transformers_ok?(pm_phy‘states,
((memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)) ∪ me
{-18} side_effect_content_unchanged((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), pm_phy‘states,
memory_read_side_effect(pm_phy‘mem))
{-19} side_effect_content_unchanged(pm_phy‘rw_addr, pm_phy‘states,
memory_write_side_effect(pm_phy‘mem))
{-20} pm′‘states = pm_phy‘states
{-21} linear_resolve_register_transformers(q′)

{1} (singleton(q′) ⊂ ((pm_phy‘other_actions ∪ memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_p

Keeping (-11 -21 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_unchanged_singleton_linear_resolve_reg_transformers.2`.
Q.E.D.

C.117.17 Linear_Memory_Properties.pm_address_in_pt_range_in_rw_addr

Terse proof for `pm_address_in_pt_range_in_rw_addr`.
`pm_address_in_pt_range_in_rw_addr:`

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy]):
is_linear_plain_memory?(pm) ∧ pm‘states(s) ⊃
(address_in_pt_range?(s, restrict [Address, Memory_Address_4G, boolean]((pm‘ro_addr ∪ pm‘

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-7 1) and hiding *,
Case splitting on union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)),
we get 2 subgoals:
`pm_address_in_pt_range_in_rw_addr.1:`

{-1} (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Ad
= restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
{1} (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘r

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_address_in_pt_range_in_rw_addr.1`.
`pm_address_in_pt_range_in_rw_addr.2:`

{-1} (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
{1} (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Ad
= restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{2} (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘r

Keeping (1) and hiding *,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_address_in_pt_range_in_rw_addr.2`.
Q.E.D.

C.117.18 Linear_Memory_Properties.address_block_in_memory_TCC1

Terse proof for `address_block_in_memory_TCC1`.

`address_block_in_memory_TCC1:`

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Address, bl: list[Byte]):
is_linear_plain_memory?(pm) ∧
(address_block(a, length(bl)) ⊆ (pm`ro_addr ∪ pm`rw_addr)) ∧ cons?(bl)
⊃ Mem?(max_linear`type_of)

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `address_block_in_memory_TCC1`.

Q.E.D.

C.117.19 Linear_Memory_Properties.address_block_in_memory

Terse proof for `address_block_in_memory`.

`address_block_in_memory:`

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Address, bl: list[Byte]):
is_linear_plain_memory?(pm) ∧
(address_block(a, length(bl)) ⊆ (pm`ro_addr ∪ pm`rw_addr)) ∧ cons?(bl)
⊃ a + length(bl) ≤ reg_size(max_linear)(type_of(a))

Repeatedly Skolemizing and flattening,

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`address_block_in_memory.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} in_memory(min_linear, max_linear)($a′$ + (length(bl′) − 1))
{-3} Mem?(type_of($a′$ + (length(bl′) − 1)))
{-4} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-5} (address_block($a′$, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-6} cons?(bl′)
{1} $a′$ + length(bl′) ≤ reg_size(max_linear)(type_of($a′$))

Expanding the definition of +,

Keeping (-2 -3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `address_block_in_memory.1`.

`address_block_in_memory.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                  real_pred(x) ∧
                    rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-3}  (address_block(a′, length(bl′)) ⊆ (pm′ro_addr ∪ pm′rw_addr))
{-4}  cons?(bl′)
-------------------------------------------------------------------------------
{1}   union(pm′ro_addr, pm′rw_addr)(a′ + (length(bl′) − 1))
{2}   a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
```

Keeping (-3 -4 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `address_block_in_memory.2`.
Q.E.D.

C.117.20 Linear_Memory_Properties.address_block_in_memory2

Terse proof for `address_block_in_memory2`.

`address_block_in_memory2:`

```
-------------------------------------------------------------------------------
{1}   ∀ (pm: Plain_Memory[Linear_memory], a: Address, bl: list[Byte]):
        is_linear_plain_memory?(pm) ∧
          (address_block(a, length(bl)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ cons?(bl)
          ⊃ 0 ≤ a‘offset
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of subset?,
Expanding the definition of member,
Instantiating the top quantifier in -2 with the terms: (a!1),
Expanding the definition of address_block,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_memory_addr,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `address_block_in_memory2`.
Q.E.D.

C.117.21 Linear_Memory_Properties.raise_fault_transformer_invariant

Terse proof for `raise_fault_transformer_invariant`.

`raise_fault_transformer_invariant:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], priv: Memory_privilege, access: Memory_access,
        present: bool, pfa: Memory_Address_4G):
       is_linear_plain_memory?(pm) ⊃
         transformer_invariant?(pm_phy`states,
                                singleton(expr_2_super(raise_fault(priv, access, present, pfa))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of raise_fault,

Rewriting using expr_composition_transformer_invariant, matching in *,

we get 3 subgoals:

`raise_fault_transformer_invariant.1:`

```
{-1}  Mem?(pfa′`type_of)
{-2}  0 ≤ pfa′`offset
{-3}  pfa′`offset < max_linear_offset
{-4}  is_linear_plain_memory?(pm′)
{1}   transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(write_data(pm_phy, address_data_type)
                                                    (CR2, pfa′))))
{2}   transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(write_data(pm_phy, address_data_type)
                                                    (CR2, pfa′)
                                                    ##
                                                    exception_result(Page_fault
                                                                     ((#reserved_bit_violation
                                                                         := FALSE,
                                                                       privilege := priv′,
                                                                       access
                                                                         := (singleton [Memory
                                                                       present
                                                                         := present′#))))))
```

Hiding formulas: 2,

Using lemma pm_states,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Using lemma unchanged_memory_invariant_invariant,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

This completes the proof of `raise_fault_transformer_invariant.1`.

`raise_fault_transformer_invariant.2:`

```
{-1}  Mem?(pfa′‘type_of)
{-2}  0 ≤ pfa′‘offset
{-3}  pfa′‘offset < max_linear_offset
{-4}  is_linear_plain_memory?(pm′)
------------------------------------------------------------------------
{1}   transformer_invariant?(pm_phy‘states,
                             singleton(expr_2_super(exception_result(Page_fault
                                                                     ((#reserved_bit_vio
                                                                         := FALSE,
                                                                         privi-
      lege := priv′,
                                                                         ac-
      cess
                                                                         := (singleton
                                                                         present
                                                                         := present′#)
{2}   transformer_invariant?(pm_phy‘states,
                             singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                    (CR2, pfa′)
                                                ##
                                                exception_result(Page_fault
                                                                     ((#reserved_bit_v
                                                                         := FALSE,
                                                                         priv-
      ilege := priv′,
                                                                         ac-
      cess
                                                                         := (singleton
                                                                         present
                                                                         := present′#
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `raise_fault_transformer_invariant.2`.

`raise_fault_transformer_invariant.3`:

```
{-1}  Mem?(pfa′`type_of)
{-2}  0 ≤ pfa′`offset
{-3}  pfa′`offset < max_linear_offset
{-4}  is_linear_plain_memory?(pm′)
------------------------------------------------------------------------
{1}   (Read ∈ (singleton[Memory_access](access′) ∪ {Read}))
{2}   transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                    (CR2, pfa′)
                                                ##
                                                exception_result(Page_fault
                                                                 ((#reserved_bit_violation
                                                                   := FALSE,
                                                                   priv-
      ilege := priv′,
                                                                   ac-
      cess
                                                                   := (singleton [Memory.
                                                                   present
                                                                   := present′#))))))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `raise_fault_transformer_invariant.3`.
Q.E.D.

C.117.22 Linear_Memory_Properties.raise_fault_unchanged_memory_invariant

Terse proof for `raise_fault_unchanged_memory_invariant`.

`raise_fault_unchanged_memory_invariant`:

```

------------------------------------------------------------------------
{1}   ∀ (pm: Plain_Memory[Linear_memory], priv: Memory_privilege, access: Mem-
      ory_access,
            present: bool, pfa: Memory_Address_4G):
        is_linear_plain_memory?(pm) ⊃
          unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                      singleton(expr_2_super(raise_fault(priv,
                                                                         ac-
      cess,
                                                                         present,
                                                                         pfa))),
                                      (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of raise_fault,
Using lemma plain_memory_unchanged_composition[Physical_memory, Unit, [bool, Address]],
we get 2 subgoals:

`raise_fault_unchanged_memory_invariant.1:`

{-1} (plain_memory?(pm_phy) ∧
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)) ∧
(∀ (s: (pm_phy‘states)): TRUE) ∧
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, address_data_type)
(CR2, pfa′))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
∧
(∀ (d: (λ (u: Unit): TRUE)):
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(exception_result
[Physical_memory,
[bool, Address]]
(Page_fault
((#reserved_bit_viol
:= FALSE,
privilege := priv′,
access
:= (singleton
present
:= present′#))
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, address_data_type)
(CR2, pfa′)
##
(λ (u: Unit):
exception_result
[Physical_memory, [bool, address]]
(Page_fault
((#reserved_bit_violation
privilege := priv′,
access
:= (singleton [Mem
present := present′
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} Mem?(pfa′‘type_of)
{-3} 0 ≤ pfa′‘offset
{-4} pfa′‘offset < max_linear_offset
{-5} is_linear_plain_memory?(pm′)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, address_data_type)
(CR2, pfa′)
##
exception_result(Page_fault
((#reserved
:= FAI
privilege := priv′,
access
:= (si
present
:= pre

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

`raise_fault_unchanged_memory_invariant.1.1:`

```
{-1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                                   (CR2, pfa′)
                                                            ##
                                                            (λ (u: Unit):
                                                                exception_result
                                                                    [Physical_memory,
                                                                      [bool, Ad-
      dress]]
                                                                   (Page_fault
                                                                     ((#reserved_bit_violation
                                                                         := FALSE,
                                                                         privi-
      lege := priv′,
                                                                         access
                                                                         := (singleton [Memory_acc
                                                                         present
                                                                         := present′#))))))),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2}  Mem?(pfa′`type_of)
{-3}  0 ≤ pfa′`offset
{-4}  pfa′`offset < max_linear_offset
{-5}  is_linear_plain_memory?(pm′)
-------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                                   (CR2, pfa′)
                                                            ##
                                                            excep-
      tion_result(Page_fault
                                                                ((#reserved_bit_violati
                                                                    := FALSE,
                                                                    priv-
      ilege := priv′,
                                                                    ac-
      cess
                                                                    := (singleton [M
                                                                    present
                                                                    := present′#)))))
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Expanding the definition of ##,

Expanding the definition of ##,

which is trivially true.

This completes the proof of `raise_fault_unchanged_memory_invariant.1.1`.

`raise_fault_unchanged_memory_invariant.1.2:`

{-1} Mem?(pfa′‘type_of)
{-2} 0 ≤ pfa′‘offset
{-3} pfa′‘offset < max_linear_offset
{-4} is_linear_plain_memory?(pm′)

{1} ∀ (*d*: (λ (*u*: Unit): TRUE)):
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(exception_result
[Physical_memory, [bool,
dress]]
(Page_fault
((#reserved_bit_violation
privi-
lege := priv′,
access
:= (singleton [Mem
present := present′
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, ad-
dress_data_type)
(CR2, pfa′)
##
excep-
tion_result(Page_fault
((#reserved
:= FAI
priv-
ilege := priv′,
ac-
cess
:= (si
present
:= pre
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `raise_fault_unchanged_memory_invariant.1.2`.

`raise_fault_unchanged_memory_invariant.1.3:`

{-1} Mem?(pfa′‘type_of)
{-2} 0 ≤ pfa′‘offset
{-3} pfa′‘offset < max_linear_offset
{-4} is_linear_plain_memory?(pm′)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, address_data_type)
(CR2, pfa′))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, address_data_type)
(CR2, pfa′)
##
exception_result(Page_fault
((#reserved_bit_violation := FALSE,
privilege := priv′,
access
:= (singleton [Mem
present
:= present′#)))))
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Hiding formulas: 2,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of linear_resolve_register_transformers,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

This completes the proof of `raise_fault_unchanged_memory_invariant.1.3`.

`raise_fault_unchanged_memory_invariant.1.4:`

```
{-1}  Mem?(pfa'‘type_of)
{-2}  0 ≤ pfa'‘offset
{-3}  pfa'‘offset < max_linear_offset
{-4}  is_linear_plain_memory?(pm')
------------------------------------------------------------------------
{1}   ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                         (CR2, pfa')
                                                  ##
                                                  excep-
      tion_result(Page_fault
                                                                  ((#reserved
                                                                      := FAI
                                                                      priv-
      ilege := priv',
                                                                      ac-
      cess
                                                                      := (sin
                                                                      present
                                                                      := pre
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
```

Rewriting using subset_reflexive, matching in *,
This completes the proof of `raise_fault_unchanged_memory_invariant.1.4`.
`raise_fault_unchanged_memory_invariant.1.5:`

```
{-1}  Mem?(pfa'‘type_of)
{-2}  0 ≤ pfa'‘offset
{-3}  pfa'‘offset < max_linear_offset
{-4}  is_linear_plain_memory?(pm')
------------------------------------------------------------------------
{1}   plain_memory?(pm_phy)
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                         (CR2, pfa')
                                                  ##
                                                  excep-
      tion_result(Page_fault
                                                                  ((#reserved
                                                                      := FAI
                                                                      priv-
      ilege := priv',
                                                                      ac-
      cess
                                                                      := (sin
                                                                      present
                                                                      := pre
                                  (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
```

Using lemma pm_plain_phy,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `raise_fault_unchanged_memory_invariant.1.5`.

`raise_fault_unchanged_memory_invariant.2`:

```
{-1}  Mem?(pfa′`type_of)
{-2}  0 ≤ pfa′`offset
{-3}  pfa′`offset < max_linear_offset
{-4}  is_linear_plain_memory?(pm′)
------------------------------------------------------------------------
{1}   (Read ∈ (singleton[Memory_access](access′) ∪ {Read}))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(write_data(pm_phy, ad-
      dress_data_type)
                                                         (CR2, pfa′)
                                                      ##
                                                      excep-
      tion_result(Page_fault
                                                                  ((#reserved_bit_violati
                                                                      := FALSE,
                                                                      priv-
      ilege := priv′,
                                                                      ac-
      cess
                                                                      := (singleton [M
                                                                      present
                                                                      := present′#)))))
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `raise_fault_unchanged_memory_invariant.2`.

Q.E.D.

C.117.23 Linear_Memory_Properties.apply_side_effects_pm_states

Terse proof for `apply_side_effects_pm_states`.

`apply_side_effects_pm_states`:

```
------------------------------------------------------------------------
{1}   ∀ (pm: Plain_Memory[Linear_memory], l: list[[Memory_Address_4G, list[Byte]]],
          f:
            [Memory_Address_4G, list[Byte] →
                 [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
        is_linear_plain_memory?(pm) ∧
         every(λ (e: [Memory_Address_4G, list[Byte]]):
                   transformer_invariant?(pm`states, singleton(expr_2_super(f(e)))))
               (l)
         ⊃ transformer_invariant?(pm`states, singleton(expr_2_super(apply_side_effects(l, f))))
```

Expanding the definition of transformer_invariant?,

Inducting on l on formula 1,

we get 2 subgoals:

`apply_side_effects_pm_states.1:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory],
          f:
            [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
       is_linear_plain_memory?(pm) ∧
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                  ∀ (s: Linear_memory[Physical_memory, pm_phy],
                     q:
                       [Linear_memory[Physical_memory, pm_phy] →
                           SuperResult[Physical_memory]]):
                    pm‘states(s) ∧ singleton(expr_2_super(f(e)))(q) ⊃
                     result_pred(pm‘states)(q(s)))
              (null)
        ⊃
        (∀ (s: Linear_memory[Physical_memory, pm_phy],
            q: [Linear_memory[Physical_memory, pm_phy] → SuperRe-
     sult[Physical_memory]]):
           pm‘states(s) ∧ singleton(expr_2_super(apply_side_effects(null, f)))(q) ⊃
            result_pred(pm‘states)(q(s)))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of apply_side_effects,

Expanding the definition of singleton,

Replacing using formula -4,

Expanding the definition of reduce,

Installing automatic rewrites from: result_pred expr_2_super expr_2_super_res ok_result

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_pm_states.1`.

`apply_side_effects_pm_states.2:`

```
{1}  ∀ (cons1_var: [Memory_Address_4G, list[Byte]],
        cons2_var: list[[Memory_Address_4G, list[Byte]]]):
       (∀ (pm: Plain_Memory[Linear_memory],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
          is_linear_plain_memory?(pm) ∧
           every(λ (e: [Memory_Address_4G, list[Byte]]):
                   ∀ (s: Linear_memory[Physical_memory, pm_phy],
                      q:
                        [Linear_memory[Physical_memory, pm_phy] →
                           SuperResult[Physical_memory]]):
                     pm‘states(s) ∧ singleton(expr_2_super(f(e)))(q) ⊃
                      result_pred(pm‘states)(q(s)))
                 (cons2_var)
           ⊃
           (∀ (s: Linear_memory[Physical_memory, pm_phy],
               q: [Linear_memory[Physical_memory, pm_phy] → SuperRe-
sult[Physical_memory]]):
              pm‘states(s) ∧
               singleton(expr_2_super(apply_side_effects(cons2_var, f)))(q)
               ⊃ result_pred(pm‘states)(q(s))))
       ⊃
       (∀ (pm: Plain_Memory[Linear_memory],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
          is_linear_plain_memory?(pm) ∧
           every(λ (e: [Memory_Address_4G, list[Byte]]):
                   ∀ (s: Linear_memory[Physical_memory, pm_phy],
                      q:
                        [Linear_memory[Physical_memory, pm_phy] →
                           SuperResult[Physical_memory]]):
                     pm‘states(s) ∧ singleton(expr_2_super(f(e)))(q) ⊃
                      result_pred(pm‘states)(q(s)))
                 (cons(cons1_var, cons2_var))
           ⊃
           (∀ (s: Linear_memory[Physical_memory, pm_phy],
               q:
                 [Linear_memory[Physical_memory, pm_phy] → SuperRe-
sult[Physical_memory]]):
              pm‘states(s) ∧
               singleton(expr_2_super(apply_side_effects(cons(cons1_var, cons2_var), f)))
                 (q)
               ⊃ result_pred(pm‘states)(q(s))))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: (s!1 expr_2_super(apply_side_effects(cons2_var!1, f!1))),
Expanding the definition of singleton,
Replacing using formula -6,
Hiding formulas: -6,
Keeping (-3 -4 -5 1) and hiding *,
Expanding the definition of apply_side_effects,
Expanding the definition of reduce,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `apply_side_effects_pm_states.2`.
Q.E.D.

C.117.24 Linear_Memory_Properties.apply_side_effects_ok

Terse proof for `apply_side_effects_ok`.
`apply_side_effects_ok:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          l: list[[Memory_Address_4G, list[Byte]]],
          f:
            [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
        is_linear_plain_memory?(pm) ∧
         pm‘states(s) ∧
          (∀ (s_1: (pm‘states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))(l))
          ∧
          every(λ (e: [Memory_Address_4G, list[Byte]]):
                    transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                (l)
         ⊃ OK?(apply_side_effects(l, f)(s))
```

Inducting on l on formula 1,
we get 2 subgoals:
`apply_side_effects_ok.1:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          f:
            [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
        is_linear_plain_memory?(pm) ∧
         pm‘states(s) ∧
          (∀ (s_1: (pm‘states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))(null))
          ∧
          every(λ (e: [Memory_Address_4G, list[Byte]]):
                    transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                (null)
         ⊃ OK?(apply_side_effects(null, f)(s))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of apply_side_effects,

Expanding the definition of ok_result,

Expanding the definition of reduce,

which is trivially true.

This completes the proof of `apply_side_effects_ok.1`.

`apply_side_effects_ok.2`:

```
{1}  ∀ (cons1_var: [Memory_Address_4G, list[Byte]],
          cons2_var: list[[Memory_Address_4G, list[Byte]]]):
        (∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
              f:
                [Memory_Address_4G, list[Byte] →
                    [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
           is_linear_plain_memory?(pm) ∧
            pm‘states(s) ∧
             (∀ (s_1: (pm‘states)):
                every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))
                    (cons2_var))
              ∧
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                        transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                    (cons2_var)
           ⊃ OK?(apply_side_effects(cons2_var, f)(s)))
         ⊃
         (∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
               f:
                 [Memory_Address_4G, list[Byte] →
                     [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
            is_linear_plain_memory?(pm) ∧
             pm‘states(s) ∧
              (∀ (s_1: (pm‘states)):
                 every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))
                     (cons(cons1_var, cons2_var)))
               ∧
               every(λ (e: [Memory_Address_4G, list[Byte]]):
                         transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                     (cons(cons1_var, cons2_var))
            ⊃ OK?(apply_side_effects(cons(cons1_var, cons2_var), f)(s)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`apply_side_effects_ok.2.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states(s′)
{-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
            (cons2_var′)
{-4}  OK?(apply_side_effects(cons2_var′, f′)(s′))
{-5}  ∀ (s1: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s1)))
            (cons(cons1_var′, cons2_var′))
{-6}  transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(cons1_var′))))
------------------------------------------------------------------------------
{1}   OK?(apply_side_effects(cons(cons1_var′, cons2_var′), f′)(s′))
```

Using lemma apply_side_effects_pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma expr_transformer_invariant_next_ok,

Installing automatic rewrites from: (has_next_state! singleton!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Expanding the definition of ##,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -7 with the terms: (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail))))))) (cons2_var!1)(s!1))),

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using ok_result_ok, matching in *,

This completes the proof of `apply_side_effects_ok.2.1`.

`apply_side_effects_ok.2.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states(s′)
{-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
            (cons2_var′)
{-4}  ∀ (s1: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s1)))
            (cons(cons1_var′, cons2_var′))
{-5}  transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(cons1_var′))))
------------------------------------------------------------------------------
{1}   ∀ (s1: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s1)))(cons2_var′)
{2}   OK?(apply_side_effects(cons(cons1_var′, cons2_var′), f′)(s′))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_ok.2.2`.

Q.E.D.

C.117.25 Linear_Memory_Properties.apply_side_effects_same_result_TCC1

Terse proof for `apply_side_effects_same_result_TCC1`.

`apply_side_effects_same_result_TCC1`:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
l: list[[Memory_Address_4G, list[Byte]]],
f:
[Memory_Address_4G, list[Byte] →
[Linear_memory[Physical_memory, pm_phy] →
ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]]):
is_linear_plain_memory?(pm) ∧
pm‘states(s) ∧
(∀ (s_1: (pm‘states)):
every(λ (e: [Memory_Address_4G, list[Byte]]): OK?($f(e)(s_1)$))(l))
∧
(∀ (s_1: (pm‘states)):
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?($f(e)(s_1)$) ⊃ data($f(e)(s_1)$) = e‘2)
(l))
∧
every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm‘states, singleton(expr_2_super($f(e)$))))
(l)
⊃
OK?[Physical_memory, list[Byte]]
(apply_side_effects[Physical_memory, pm_phy](l, f)(s))

Repeatedly Skolemizing and flattening,

Using lemma apply_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result_TCC1`.

Q.E.D.

C.117.26 Linear_Memory_Properties.apply_side_effects_same_result

Terse proof for `apply_side_effects_same_result`.

`apply_side_effects_same_result:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          l: list[[Memory_Address_4G, list[Byte]]],
          f:
            [Memory_Address_4G, list[Byte] →
                 [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
       is_linear_plain_memory?(pm) ∧
        pm‘states(s) ∧
         (∀ (s1: (pm‘states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))(l))
          ∧
          (∀ (s1: (pm‘states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                        OK?(f(e)(s1)) ⊃ data(f(e)(s1)) = e‘2)
                    (l))
           ∧
           every(λ (e: [Memory_Address_4G, list[Byte]]):
                     transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                 (l)
        ⊃
        data(apply_side_effects(l, f)(s)) =
         reduce(null,
                λ (e: [Address, list[Byte]], tail: list[Byte]):
                  LET head = e‘2 IN append(head, tail))
               (l)
```

Inducting on l on formula 1,

we get 4 subgoals:

`apply_side_effects_same_result.1`:

```
{-1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
        is_linear_plain_memory?(pm) ∧
         pm`states(s) ∧
          (∀ (s1: (pm`states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))(l'))
           ∧
           (∀ (s1: (pm`states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                        OK?(f(e)(s1)) ⊃ data(f(e)(s1)) = e`2)
                    (l'))
            ∧
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                      transformer_invariant?(pm`states, singleton(expr_2_super(f(e)))))
                  (l')
         ⊃
         data(apply_side_effects(l', f)(s)) =
          reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                 (l')
-------------------------------------------------------------------------------
{1}   ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
        is_linear_plain_memory?(pm) ∧
         pm`states(s) ∧
          (∀ (s1: (pm`states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))(l'))
           ∧
           (∀ (s1: (pm`states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                        OK?(f(e)(s1)) ⊃ data(f(e)(s1)) = e`2)
                    (l'))
            ∧
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                      transformer_invariant?(pm`states, singleton(expr_2_super(f(e)))))
                  (l')
         ⊃
         data(apply_side_effects(l', f)(s)) =
          reduce(null,
                 λ (e: [Address, list[Byte]], tail: list[Byte]):
                   LET head = e`2 IN append(head, tail))
                 (l')
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.1`.

`apply_side_effects_same_result.2:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          f:
            [Memory_Address_4G, list[Byte] →
                 [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
        is_linear_plain_memory?(pm) ∧
         pm‘states(s) ∧
          (∀ (s1: (pm‘states)):
              every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s1)))(null))
           ∧
           (∀ (s1: (pm‘states)):
               every(λ (e: [Memory_Address_4G, list[Byte]]):
                          OK?(f(e)(s1)) ⊃ data(f(e)(s1)) = e‘2)
                     (null))
            ∧
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                       transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                  (null)
         ⊃
         data(apply_side_effects(null, f)(s)) =
          reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
                 (null)
```

Repeatedly Skolemizing and flattening,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of `apply_side_effects_same_result.2`.

apply_side_effects_same_result.3:

```
{1}  ∀ (cons1_var: [Memory_Address_4G, list[Byte]],
        cons2_var: list[[Memory_Address_4G, list[Byte]]]):
       (∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
          is_linear_plain_memory?(pm) ∧
           pm‘states(s) ∧
            (∀ (s_1: (pm‘states)):
               every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))
                 (cons2_var))
             ∧
             (∀ (s_1: (pm‘states)):
                every(λ (e: [Memory_Address_4G, list[Byte]]):
                        OK?(f(e)(s_1)) ⊃ data(f(e)(s_1)) = e‘2)
                  (cons2_var))
              ∧
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                      transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
                (cons2_var)
          ⊃
          data(apply_side_effects(cons2_var, f)(s)) =
           reduce(null,
                  λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
             (cons2_var))
       ⊃
       (∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
          is_linear_plain_memory?(pm) ∧
           pm‘states(s) ∧
            (∀ (s_1: (pm‘states)):
               every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))
                 (cons(cons1_var, cons2_var)))
             ∧
             (∀ (s_1: (pm‘states)):
                every(λ (e: [Memory_Address_4G, list[Byte]]):
                        OK?(f(e)(s_1)) ⊃ data(f(e)(s_1)) = e‘2)
                  (cons(cons1_var, cons2_var)))
              ∧
              every(λ (e: [Memory_Address_4G, list[Byte]]):
                      transformer_invariant?(pm‘states, single-
    ton(expr_2_super(f(e)))))
                (cons(cons1_var, cons2_var))
          ⊃
          data(apply_side_effects(cons(cons1_var, cons2_var), f)(s)) =
           reduce(null,
                  λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
             (cons(cons1_var, cons2_var)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Case splitting on FORALL (s1: (pm!1‘states)): every(LAMBDA (e: [Memory_Address_4G, list[Byte]]): OK?(f!1(e)(s1))) (cons2_var!1),

we get 2 subgoals:

`apply_side_effects_same_result.3.1:`

{-1} $\forall\ (s_1\colon (\mathrm{pm}'\text{‘states})):$
$\quad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):\ \mathrm{OK?}(f'(e)(s_1)))(\mathrm{cons2\_var}')$

{-2} $\mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}')\ \wedge$
$\quad \mathrm{pm}'\text{‘states}(s')\ \wedge$
$\quad (\forall\ (s_1\colon (\mathrm{pm}'\text{‘states})):$
$\qquad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):\ \mathrm{OK?}(f'(e)(s_1)))$
$\qquad\quad (\mathrm{cons2\_var}'))$
$\quad \wedge$
$\quad (\forall\ (s_1\colon (\mathrm{pm}'\text{‘states})):$
$\qquad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):$
$\qquad\quad \mathrm{OK?}(f'(e)(s_1))\ \supset\ \mathrm{data}(f'(e)(s_1))\ =\ e\text{‘}2)$
$\qquad\quad (\mathrm{cons2\_var}'))$
$\quad \wedge$
$\quad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):$
$\qquad\quad \mathrm{transformer\_invariant?}(\mathrm{pm}'\text{‘states},\ \mathrm{singleton}(\mathrm{expr\_2\_super}(f'(e)))))$
$\qquad (\mathrm{cons2\_var}')$
$\supset$
$\mathrm{data}(\mathrm{apply\_side\_effects}(\mathrm{cons2\_var}',\ f')(s'))\ =$
$\quad \mathrm{reduce}(\mathrm{null},\ \lambda\ (e\colon [\mathrm{Address},\ \mathrm{list}[\mathrm{Byte}]],\ \mathrm{tail}\colon \mathrm{list}[\mathrm{Byte}]):\ \mathrm{append}(e\text{‘}2,\ \mathrm{tail}))$
$\qquad (\mathrm{cons2\_var}')$

{-3} $\mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}')$

{-4} $\mathrm{pm}'\text{‘states}(s')$

{-5} $\forall\ (s_1\colon (\mathrm{pm}'\text{‘states})):$
$\quad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):\ \mathrm{OK?}(f'(e)(s_1)))$
$\qquad (\mathrm{cons}(\mathrm{cons1\_var}',\ \mathrm{cons2\_var}'))$

{-6} $\forall\ (s_1\colon (\mathrm{pm}'\text{‘states})):$
$\quad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):$
$\qquad\quad \mathrm{OK?}(f'(e)(s_1))\ \supset\ \mathrm{data}(f'(e)(s_1))\ =\ e\text{‘}2)$
$\qquad (\mathrm{cons}(\mathrm{cons1\_var}',\ \mathrm{cons2\_var}'))$

{-7} $\mathrm{transformer\_invariant?}(\mathrm{pm}'\text{‘states},\ \mathrm{singleton}(\mathrm{expr\_2\_super}(f'(\mathrm{cons1\_var}'))))\ \wedge$
$\quad \mathrm{every}(\lambda\ (e\colon [\mathrm{Memory\_Address\_4G},\ \mathrm{list}[\mathrm{Byte}]]):$
$\qquad\quad \mathrm{transformer\_invariant?}(\mathrm{pm}'\text{‘states},\ \mathrm{singleton}(\mathrm{expr\_2\_super}(f'(e)))))$
$\qquad (\mathrm{cons2\_var}')$

{1} $\mathrm{data}(\mathrm{apply\_side\_effects}(\mathrm{cons}(\mathrm{cons1\_var}',\ \mathrm{cons2\_var}'),\ f')(s'))\ =$
$\quad \mathrm{reduce}(\mathrm{null},\ \lambda\ (e\colon [\mathrm{Address},\ \mathrm{list}[\mathrm{Byte}]],\ \mathrm{tail}\colon \mathrm{list}[\mathrm{Byte}]):\ \mathrm{append}(e\text{‘}2,\ \mathrm{tail}))$
$\qquad (\mathrm{cons}(\mathrm{cons1\_var}',\ \mathrm{cons2\_var}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

apply_side_effects_same_result.3.1.1:

```
{-1}  ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s₁)))(cons2_var′)
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
          (cons2_var′)
{-5}  data(apply_side_effects(cons2_var′, f′)(s′)) =
       reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
           (cons2_var′)
{-6}  ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s₁)))
           (cons(cons1_var′, cons2_var′))
{-7}  ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(f′(e)(s₁)) ⊃ data(f′(e)(s₁)) = e‘2)
           (cons(cons1_var′, cons2_var′))
{-8}  transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(cons1_var′))))
-------------------------------------------------------------------------------
 {1}  data(apply_side_effects(cons(cons1_var′, cons2_var′), f′)(s′)) =
       reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
           (cons(cons1_var′, cons2_var′))
```

Using lemma apply_side_effects_ok,

Using lemma apply_side_effects_pm_states,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Expanding the definition of ##,

Expanding the definition of ##,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of every,

Rewriting using forall_and, matching in *,

Rewriting using forall_and, matching in *,

Applying disjunctive simplification to flatten sequent,

Replacing using formula -9,

Expanding the definition of apply_side_effects,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on pm!1‘states (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory [Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail))))))) (cons2_var!1)(s!1))),

we get 2 subgoals:

apply_side_effects_same_result.3.1.1.1:

{-1} pm′‘states
(state(reduce(ok_result(null),
λ (e: [Memory_Address_4G, list[Byte]],
r:
[Linear_memory[Physical_memory, pm_phy] →
ExprResult
[Linear_memory[Physical_memory, pm_phy], list[By
(r ##
(λ (tail: list[Byte]):
(f′(e) ##
(λ (head: list[Byte]):
ok_result(append(head, tail)))))))
(cons2_var′)(s′)))

{-2} transformer_invariant?(pm′‘states,
singleton(expr_2_super(λ (s:
Lin-
ear_memory[Physical_memory, pm_phy]):
reduce(ok_result(null),
λ (e:
[Memory_Address_4G
list[Byte]],
r:
[Linear_memory
[Physical_memory
ExprResult
[Linear_memory
[Physical_memo
list[Byte]]]]):
(r
##
(λ
(tail: list[Byte]):
(f′(e)
##
(λ
(head: list[Byt
ok_result
(append
(head, tail))
(cons2_var′)(s))))

{-3} OK?(reduce(ok_result(null),
λ (e: [Memory_Address_4G, list[Byte]],
r:
[Linear_memory[Physical_memory, pm_phy] →
ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]
(r ##
(λ (tail: list[Byte]):
(f′(e) ##
(λ (head: list[Byte]): ok_result(append(head, tail)))))))
(cons2_var′)(s′))

{-4} is_linear_plain_memory?(pm′)

{-5} pm′‘states(s′)

{-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
(cons2_var′)

{-7} data(reduce(ok_result(null),
λ (e: [Memory_Address_4G, list[Byte]],
r:
[Linear_memory[Physical_memory, pm_phy] →
ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte
(r ##
(λ (tail: list[Byte]):
(f′(e) ##

Instantiating the top quantifier in -8 with the terms: (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail))))))) (cons2_var!1)(s!1))),

we get 2 subgoals:

```
apply_side_effects_same_result.3.1.1.1.1:

{-1}  pm'‘states
        (state(reduce(ok_result(null),
                      λ (e: [Memory_Address_4G, list[Byte]],
                         r:
                           [Linear_memory[Physical_memory, pm_phy] →
                              ExprResult
                                [Linear_memory[Physical_memory, pm_phy], list[By
                       (r ##
                          (λ (tail: list[Byte]):
                             (f'(e) ##
                                (λ (head: list[Byte]):
                                   ok_result(append(head, tail)))))))
              (cons2_var')(s')))
{-2}  transformer_invariant?(pm'‘states,
                             singleton(expr_2_super(λ (s:
                                                        Lin-
      ear_memory[Physical_memory, pm_phy]):
                                                      reduce(ok_result(null),
                                                             λ (e:
                                                                [Memory_Address_4G
                                                                  list[Byte]],
                                                                r:
                                                                [Linear_memory
                                                                  [Physical_memory
                                                                  ExprResult
                                                                  [Linear_memory
                                                                    [Physical_memo
                                                                    list[Byte]]]]):
                                                               (r
                                                                 ##
                                                                 (λ
                                                                   (tail: list[Byte]):
                                                                   (f'(e)
                                                                     ##
                                                                     (λ
                                                                       (head: list[Byt
                                                                       ok_result
                                                                       (append
                                                                         (head, tail))
                                                             (cons2_var')(s))))
{-3}  OK?(reduce(ok_result(null),
                 λ (e: [Memory_Address_4G, list[Byte]],
                    r:
                      [Linear_memory[Physical_memory, pm_phy] →
                         ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]
                   (r ##
                      (λ (tail: list[Byte]):
                         (f'(e) ##
                            (λ (head: list[Byte]): ok_result(append(head, tail)))))))
               (cons2_var')(s'))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'‘states(s')
{-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm'‘states, singleton(expr_2_super(f'(e)))))
        (cons2_var')
{-7}  data(reduce(ok_result(null),
                  λ (e: [Memory_Address_4G, list[Byte]],
                     r:
                       [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte
                    (r ##
                       (λ (tail: list[Byte]):
                          (f'(e) ##
```

Instantiating the top quantifier in -10 with the terms: (state(reduce(ok_result(null), LAMBDA (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] -> ExprResult [Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (LAMBDA (tail: list[Byte]): (f!1(e) ## (LAMBDA (head: list[Byte]): ok_result(append(head, tail))))))) (cons2_var!1)(s!1))),

we get 2 subgoals:

```
apply_side_effects_same_result.3.1.1.1.1.1:

{-1}  pm'‘states
        (state(reduce(ok_result(null),
                      λ (e: [Memory_Address_4G, list[Byte]],
                         r:
                           [Linear_memory[Physical_memory, pm_phy] →
                              ExprResult
                                [Linear_memory[Physical_memory, pm_phy], list[By
                       (r ##
                          (λ (tail: list[Byte]):
                             (f'(e) ##
                                (λ (head: list[Byte]):
                                   ok_result(append(head, tail)))))))
                    (cons2_var')(s')))
{-2}  transformer_invariant?(pm'‘states,
                             singleton(expr_2_super(λ (s:
                                                        Lin-
      ear_memory[Physical_memory, pm_phy]):
                                                      reduce(ok_result(null),
                                                             λ (e:
                                                                [Memory_Address_4G
                                                                  list[Byte]],
                                                                r:
                                                                [Linear_memory
                                                                  [Physical_memory
                                                                  ExprResult
                                                                  [Linear_memory
                                                                    [Physical_memo
                                                                    list[Byte]]]]):
                                                               (r
                                                                 ##
                                                                 (λ
                                                                   (tail: list[Byte]):
                                                                   (f'(e)
                                                                     ##
                                                                     (λ
                                                                       (head: list[Byt
                                                                       ok_result
                                                                       (append
                                                                         (head, tail))
                                                             (cons2_var')(s))))
{-3}  OK?(reduce(ok_result(null),
                 λ (e: [Memory_Address_4G, list[Byte]],
                    r:
                      [Linear_memory[Physical_memory, pm_phy] →
                         ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]
                   (r ##
                      (λ (tail: list[Byte]):
                         (f'(e) ##
                            (λ (head: list[Byte]): ok_result(append(head, tail)))))))
                (cons2_var')(s'))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'‘states(s')
{-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm'‘states, singleton(expr_2_super(f'(e)))))
        (cons2_var')
{-7}  data(reduce(ok_result(null),
                  λ (e: [Memory_Address_4G, list[Byte]],
                     r:
                       [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte
                    (r ##
                       (λ (tail: list[Byte]):
                          (f'(e) ##
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using ok_result_data, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.1.1.1.1.1`.

```
apply_side_effects_same_result.3.1.1.1.1.2:

{-1}  pm'‘states
        (state(reduce(ok_result(null),
                      λ (e: [Memory_Address_4G, list[Byte]],
                         r:
                           [Linear_memory[Physical_memory, pm_phy] →
                              ExprResult
                                [Linear_memory[Physical_memory, pm_phy], list[By
                       (r ##
                          (λ (tail: list[Byte]):
                             (f'(e) ##
                                (λ (head: list[Byte]):
                                   ok_result(append(head, tail)))))))
                      (cons2_var')(s')))
{-2}  transformer_invariant?(pm'‘states,
                             singleton(expr_2_super(λ (s:
                                                        Lin-
      ear_memory[Physical_memory, pm_phy]):
                                                      reduce(ok_result(null),
                                                             λ (e:
                                                                [Memory_Address_4G
                                                                  list[Byte]],
                                                                r:
                                                                [Linear_memory
                                                                  [Physical_memory
                                                                  ExprResult
                                                                  [Linear_memory
                                                                    [Physical_memo
                                                                    list[Byte]]]]):
                                                               (r
                                                                 ##
                                                                 (λ
                                                                   (tail: list[Byte]):
                                                                   (f'(e)
                                                                     ##
                                                                     (λ
                                                                       (head: list[Byt
                                                                       ok_result
                                                                       (append
                                                                         (head, tail))
                                                             (cons2_var')(s))))
{-3}  OK?(reduce(ok_result(null),
                 λ (e: [Memory_Address_4G, list[Byte]],
                    r:
                      [Linear_memory[Physical_memory, pm_phy] →
                         ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]
                   (r ##
                      (λ (tail: list[Byte]):
                         (f'(e) ##
                            (λ (head: list[Byte]): ok_result(append(head, tail)))))))
                 (cons2_var')(s'))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'‘states(s')
{-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm'‘states, singleton(expr_2_super(f'(e)))))
        (cons2_var')
{-7}  data(reduce(ok_result(null),
                  λ (e: [Memory_Address_4G, list[Byte]],
                     r:
                       [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte
                    (r ##
                       (λ (tail: list[Byte]):
                          (f'(e) ##
```

which is trivially true.

This completes the proof of `apply_side_effects_same_result.3.1.1.1.1.2`.

```
apply_side_effects_same_result.3.1.1.1.2:

{-1}  pm'‘states
        (state(reduce(ok_result(null),
                      λ (e: [Memory_Address_4G, list[Byte]],
                         r:
                           [Linear_memory[Physical_memory, pm_phy] →
                              ExprResult
                                [Linear_memory[Physical_memory, pm_phy], list[By
                       (r ##
                          (λ (tail: list[Byte]):
                             (f'(e) ##
                                (λ (head: list[Byte]):
                                   ok_result(append(head, tail)))))))
                    (cons2_var')(s')))
{-2}  transformer_invariant?(pm'‘states,
                             singleton(expr_2_super(λ (s:
                                                        Lin-
      ear_memory[Physical_memory, pm_phy]):
                                                      reduce(ok_result(null),
                                                             λ (e:
                                                                [Memory_Address_4G
                                                                  list[Byte]],
                                                                r:
                                                                [Linear_memory
                                                                  [Physical_memory
                                                                  ExprResult
                                                                  [Linear_memory
                                                                    [Physical_memo
                                                                    list[Byte]]]]):
                                                               (r
                                                                 ##
                                                                 (λ
                                                                   (tail: list[Byte]):
                                                                   (f'(e)
                                                                     ##
                                                                     (λ
                                                                       (head: list[Byt
                                                                       ok_result
                                                                       (append
                                                                         (head, tail))
                                                             (cons2_var')(s))))
{-3}  OK?(reduce(ok_result(null),
                 λ (e: [Memory_Address_4G, list[Byte]],
                    r:
                      [Linear_memory[Physical_memory, pm_phy] →
                         ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]
                   (r ##
                      (λ (tail: list[Byte]):
                         (f'(e) ##
                            (λ (head: list[Byte]): ok_result(append(head, tail)))))))
                (cons2_var')(s'))
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'‘states(s')
{-6}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm'‘states, singleton(expr_2_super(f'(e)))))
        (cons2_var')
{-7}  data(reduce(ok_result(null),
                  λ (e: [Memory_Address_4G, list[Byte]],
                     r:
                       [Linear_memory[Physical_memory, pm_phy] →
                          ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte
                    (r ##
                       (λ (tail: list[Byte]):
                          (f'(e) ##
```

which is trivially true.

This completes the proof of `apply_side_effects_same_result.3.1.1.1.2`.

apply_side_effects_same_result.3.1.1.2:

{-1} transformer_invariant?(pm'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s))))

{-2} OK?(reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s'))

{-3} is_linear_plain_memory?(pm')

{-4} pm'states(s')

{-5} every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(f'(e)))))(cons2_var')

{-6} data(reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s'))
=
reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))(cons2_var')

{-7} ∀ (x: (pm'states)): OK?(f'(cons1_var')(x))

{-8} ∀ (x: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)))(cons2_var')

{-9} ∀ (x: (pm'states)): OK?(f'(cons1_var')(x)) ⊃ data(f'(cons1_var')(x)) = cons1_var'2

{-10} ∀ (x: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)) ⊃ data(f'(e)(x)) = e'2)

Hiding formulas: 2,

Using lemma expr_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

apply_side_effects_same_result.3.1.1.2.1:

{-1} transformer_invariant?(pm'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s))))

{-2} pm'states(s')

{-3} OK?(reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s'))

{-4} is_linear_plain_memory?(pm')

{-5} every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(f'(e)))))(cons2_var')

{-6} data(reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s'))
=
reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))(cons2_var')

{-7} ∀ (x: (pm'states)): OK?(f'(cons1_var')(x))

{-8} ∀ (x: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)))(cons2_var')

{-9} ∀ (x: (pm'states)): OK?(f'(cons1_var')(x)) ⊃ data(f'(cons1_var')(x)) = cons1_var'2

{-10} ∀ (x: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)) ⊃ data(f'(e)(x)) = e'2)

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of `apply_side_effects_same_result.3.1.1.2.1`.

apply_side_effects_same_result.3.1.1.2.2:

{-1} transformer_invariant?(pm'states, singleton(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]): reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s))))

{-2} pm'states(s')

{-3} OK?(reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s'))

{-4} is_linear_plain_memory?(pm')

{-5} every(λ (e: [Memory_Address_4G, list[Byte]]): transformer_invariant?(pm'states, singleton(expr_2_super(f'(e)))))(cons2_var')

{-6} data(reduce(ok_result(null), λ (e: [Memory_Address_4G, list[Byte]], r: [Linear_memory[Physical_memory, pm_phy] → ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]): (r ## (λ (tail: list[Byte]): (f'(e) ## (λ (head: list[Byte]): ok_result(append(head, tail)))))))(cons2_var')(s'))
=
reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e'2, tail))(cons2_var')

{-7} ∀ (x: (pm'states)): OK?(f'(cons1_var')(x))

{-8} ∀ (x: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)))(cons2_var')

{-9} ∀ (x: (pm'states)): OK?(f'(cons1_var')(x)) ⊃ data(f'(cons1_var')(x)) = cons1_var'2

{-10} ∀ (x: (pm'states)): every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f'(e)(x)) ⊃ data(f'(e)(x)) = e'2)

Expanding the definition of singleton,

Applying decompose-equality,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.1.1.2.2`.

`apply_side_effects_same_result.3.1.2:`

```
{-1}  ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s₁)))(cons2_var′)
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
          (cons2_var′)
{-5}  ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s₁)))
            (cons(cons1_var′, cons2_var′))
{-6}  ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(f′(e)(s₁)) ⊃ data(f′(e)(s₁)) = e‘2)
            (cons(cons1_var′, cons2_var′))
{-7}  transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(cons1_var′))))
------------------------------------------------------------------------------
{1}   ∀ (s₁: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(f′(e)(s₁)) ⊃ data(f′(e)(s₁)) = e‘2)
            (cons2_var′)
{2}   data(apply_side_effects(cons(cons1_var′, cons2_var′), f′)(s′)) =
       reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
            (cons(cons1_var′, cons2_var′))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.1.2`.

`apply_side_effects_same_result.3.2:`

```
{-1}  is_linear_plain_memory?(pm′) ∧
       pm′‘states(s′) ∧
        (∀ (s1: (pm′‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s1)))
                  (cons2_var′))
         ∧
         (∀ (s1: (pm′‘states)):
             every(λ (e: [Memory_Address_4G, list[Byte]]):
                       OK?(f′(e)(s1)) ⊃ data(f′(e)(s1)) = e‘2)
                   (cons2_var′))
          ∧
          every(λ (e: [Memory_Address_4G, list[Byte]]):
                    transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
                (cons2_var′)
       ⊃
       data(apply_side_effects(cons2_var′, f′)(s′)) =
        reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
              (cons2_var′)
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  ∀ (s1: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s1)))
              (cons(cons1_var′, cons2_var′))
{-5}  ∀ (s1: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                  OK?(f′(e)(s1)) ⊃ data(f′(e)(s1)) = e‘2)
              (cons(cons1_var′, cons2_var′))
{-6}  transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(cons1_var′)))) ∧
       every(λ (e: [Memory_Address_4G, list[Byte]]):
                 transformer_invariant?(pm′‘states, singleton(expr_2_super(f′(e)))))
             (cons2_var′)
-------------------------------------------------------------------------------
{1}   ∀ (s1: (pm′‘states)):
        every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f′(e)(s1)))(cons2_var′)
{2}   data(apply_side_effects(cons(cons1_var′, cons2_var′), f′)(s′)) =
       reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
             (cons(cons1_var′, cons2_var′))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.3.2`.

apply_side_effects_same_result.4:

```
{1}  ∀ (l: list[[Memory_Address_4G, list[Byte]]],
        pm: Plain_Memory[Linear_memory[Physical_memory, pm_phy]],
        s: Linear_memory[Physical_memory, pm_phy],
        f:
          [Memory_Address_4G, list[Byte] →
              [Linear_memory[Physical_memory, pm_phy] →
                  ExprResult[Linear_memory[Physical_memory, pm_phy], list[Byte]]]]):
       is_linear_plain_memory?(pm) ∧
        pm‘states(s) ∧
         (∀ (s_1: (pm‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))(l))
         ∧
         (∀ (s_1: (pm‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                    OK?(f(e)(s_1)) ⊃ data(f(e)(s_1)) = e‘2)
                (l))
          ∧
          every(λ (e: [Memory_Address_4G, list[Byte]]):
                  transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
              (l)
        ⊃
        OK?[Physical_memory, list[Byte]]
          (apply_side_effects[Physical_memory, pm_phy](l, f)(s))
{2}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
        f:
          [Memory_Address_4G, list[Byte] →
              [Linear_memory → ExprResult[Linear_memory, list[Byte]]]]):
       is_linear_plain_memory?(pm) ∧
        pm‘states(s) ∧
         (∀ (s_1: (pm‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]): OK?(f(e)(s_1)))(l′))
         ∧
         (∀ (s_1: (pm‘states)):
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                    OK?(f(e)(s_1)) ⊃ data(f(e)(s_1)) = e‘2)
                (l′))
          ∧
          every(λ (e: [Memory_Address_4G, list[Byte]]):
                  transformer_invariant?(pm‘states, singleton(expr_2_super(f(e)))))
              (l′)
        ⊃
        data(apply_side_effects(l′, f)(s)) =
         reduce(null,
                λ (e: [Address, list[Byte]], tail: list[Byte]):
                  LET head = e‘2 IN append(head, tail))
              (l′)
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma apply_side_effects_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `apply_side_effects_same_result.4`.

Q.E.D.

C.117.27 Linear_Memory_Properties.apply_side_effects_unchanged

Terse proof for `apply_side_effects_unchanged`.

`apply_side_effects_unchanged:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], l: list[[Memory_Address_4G, list[Byte]]],
        f:
          [Memory_Address_4G, list[Byte] →
              [Linear_memory → ExprResult[Linear_memory, list[Byte]]]],
        addresses: PRED[Memory_Address_4G]):
     is_linear_plain_memory?(pm) ∧
      (addresses ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_
       ∧
       every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                            singleton(expr_2_super(f(e))),
                                            extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                                  (addresses)))
             (l)
       ⊃
       unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                   singleton(expr_2_super(apply_side_effects(l, f))),
                                   extend[Address, Mem-
ory_Address_4G, bool, FALSE](addresses))
```

Inducting on l on formula 1,

we get 2 subgoals:

`apply_side_effects_unchanged.1`:

```
{1}  ∀ (pm: Plain_Memory[Linear_memory],
        f:
          [Memory_Address_4G, list[Byte] →
              [Linear_memory → ExprResult[Linear_memory, list[Byte]]]],
        addresses: PRED[Memory_Address_4G]):
      is_linear_plain_memory?(pm) ∧
       (addresses ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
        ∧
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                            singleton(expr_2_super(f(e))),
                                            extend[Address, Mem-
    ory_Address_4G, bool, FALSE]
                                                  (addresses)))
              (null)
      ⊃
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(apply_side_effects(null, f))),
                                  extend[Address, Mem-
    ory_Address_4G, bool, FALSE](addresses))
```

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `apply_side_effects_unchanged.1`.

`apply_side_effects_unchanged.2:`

```
{1}  ∀ (cons1_var: [Memory_Address_4G, list[Byte]],
        cons2_var: list[[Memory_Address_4G, list[Byte]]]):
       (∀ (pm: Plain_Memory[Linear_memory],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]],
           addresses: PRED[Memory_Address_4G]):
          is_linear_plain_memory?(pm) ∧
           (addresses ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy
            ∧
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                    unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                                singleton(expr_2_super(f(e))),
                                                extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                                       (addresses)))
                  (cons2_var)
          ⊃
          unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                      singleton(expr_2_super(apply_side_effects
                                                             (cons2_var, f))),
                                      extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                           (addresses)))
       ⊃
       (∀ (pm: Plain_Memory[Linear_memory],
           f:
             [Memory_Address_4G, list[Byte] →
                [Linear_memory → ExprResult[Linear_memory, list[Byte]]]],
           addresses: PRED[Memory_Address_4G]):
          is_linear_plain_memory?(pm) ∧
           (addresses ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_pl
            ∧
            every(λ (e: [Memory_Address_4G, list[Byte]]):
                    unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                                singleton(expr_2_super(f(e))),
                                                extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                                       (addresses)))
                  (cons(cons1_var, cons2_var))
          ⊃
          unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                      singleton(expr_2_super(apply_side_effects
                                                             (cons(cons1_var, cons2_v
                                                               f))),
                                      extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                           (addresses)))
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of apply_side_effects,

Expanding the definition of reduce,

Using lemma pm_plain_phy,

Using lemma pm_states,

Using lemma plain_memory_unchanged_composition[Linear_memory, list[Byte], list[Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

apply_side_effects_unchanged.2.1:

{-1} plain_memory?(pm_phy)
{-2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(reduce(ok_result(null),
λ
(e:
[Memory_Address_4G
list[Byte]],
r:
[Linear_memory
[Physical_memory,
ExprResult
[Linear_memory
[Physical_memory
list[Byte]]]):
(r
##
(λ
(tail: list[Byte]):
(f′(e)
##
(λ
(head: list[By
ok_result
(append
(head, tail))
(cons2_var′)
##
(λ (tail_1: list[Byte]):
(f′(cons1_var′)
##
(λ
(head_1: list[Byte]):
ok_result
(append
(head_1, tail_1))))))
extend[Address, Memory_Address_4G, bool, FALSE](addresse
{-3} is_linear_plain_memory?(pm′)
{-4} pm′‘states = pm_phy‘states
{-5} (addresses′ ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_ad
{-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(f′(e))),
extend[Address, Mem-
ory_Address_4G, bool, FALSE]
(addresses′)))
(cons2_var′)
{-7} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(apply_side_effects(cons2_var′, f′))),
extend[Address, Memory_Address_4G, bool, FALSE](addresse
{-8} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(f′(cons1_var′))),
extend[Address, Memory_Address_4G, bool, FALSE](addresse

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_p
(reduce(ok_result(null),
λ
(e:
[Memory_Address
list[Byte]],
r:

Expanding the definition of expr_2_super,

which is trivially true.

This completes the proof of `apply_side_effects_unchanged.2.1`.

apply_side_effects_unchanged.2.2:

```
{-1}  plain_memory?(pm_phy)
{-2}  is_linear_plain_memory?(pm′)
{-3}  pm′‘states = pm_phy‘states
{-4}  (addresses′ ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_ad
{-5}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                            singleton(expr_2_super(f′(e))),
                                            extend[Address, Mem-
      ory_Address_4G, bool, FALSE]
                                                   (addresses′)))
            (cons2_var′)
{-6}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(apply_side_effects(cons2_var′, f′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresse
{-7}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(f′(cons1_var′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresse
------------------------------------------------------------------------------------------
{1}   ∀ (d: (λ (l: list[Byte]): TRUE)):
        unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                    singleton(expr_2_super((f′(cons1_var′) ##
                                                            (λ
                                                              (head_1: list[Byte]):
                                                              ok_result
                                                              (append(head_1, d))))))),
                                    extend[Address, Memory_Address_4G, bool, FALSE](addres
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(λ (s:
                                                            Linear_memory
                                                            [Physical_memory, pm_p
                                                          (reduce(ok_result(null),
                                                                  λ
                                                                  (e:
                                                                    [Memory_Address
                                                                      list[Byte]],
                                                                   r:
                                                                    [Linear_memory
                                                                      [Physical_mem
                                                                        pm_phy] →
                                                                      Ex-
      prResult
                                                                    [Linear_memor
                                                                      [Physical_me
                                                                        pm_phy],
                                                                      list[Byte]]])
                                                                  (r
                                                                    ##
                                                                    (λ
                                                                      (tail: list[Byte
                                                                      (f′(e)
                                                                        ##
                                                                        (λ
                                                                          (head: lis
                                                                          ok_result
                                                                          (append
                                                                            (head,
                                                                              tail)))
                                                                (cons2_var′)
                                                          ##
                                                          (λ
                                                            (tail_1: list[Byte]):
                                                            (f′(cons1_var′)
                                                              ##
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_unchanged_composition[Linear_memory, list[Byte], list[Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`apply_side_effects_unchanged.2.2.1:`

```
{-1}  plain_memory?(pm_phy)
{-2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(f'(cons1_var'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{-3}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (d')
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'`states = pm_phy`states
{-6}  (addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy`ro_addr ∪ pm_phy`rw_addr)))
{-7}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                          singleton(expr_2_super(f'(e))),
                                          extend[Address, Mem-
ory_Address_4G, bool, FALSE]
                                                 (addresses')))
            (cons2_var')
{-8}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(apply_side_effects(cons2_var', f'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
------------------------------------------------------------------------------
{1}   ∀ (d: (λ (l: list[Byte]): TRUE)):
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(ok_result(append(d, d')))),
                                    extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super((f'(cons1_var') ##
                                                          (λ (head_1: list[Byte]):
                                                             ok_result
                                                             (append(head_1, d')))))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
```

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `apply_side_effects_unchanged.2.2.1`.

`apply_side_effects_unchanged.2.2.2:`

```
{-1}  plain_memory?(pm_phy)
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(f′(cons1_var′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresse
{-3}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
      (d′)
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′‘states = pm_phy‘states
{-6}  (addresses′ ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_ad
{-7}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                          singleton(expr_2_super(f′(e))),
                                          extend[Address, Mem-
      ory_Address_4G, bool, FALSE]
                                                 (addresses′)))
      (cons2_var′)
{-8}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(apply_side_effects(cons2_var′, f′))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresse
-------------------------------------------------------------------------------------------
{1}   (extend[Address, Memory_Address_4G, bool, FALSE](addresses′) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘r
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super((f′(cons1_var′) ##
                                                          (λ (head_1: list[Byte]):
                                                             ok_result
                                                             (append(head_1, d′))))))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresse
```

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `apply_side_effects_unchanged.2.2.2`.

apply_side_effects_unchanged.2.3:

{-1} plain_memory?(pm_phy)
{-2} is_linear_plain_memory?(pm′)
{-3} pm′`states = pm_phy`states
{-4} (addresses′ ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy`ro_addr ∪ pm_phy`rw_addr)))
{-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(f′(e))),
extend[Address, Memory_Address_4G, bool, FALSE]
(addresses′)))
(cons2_var′)
{-6} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(apply_side_effects(cons2_var′, f′))),
extend[Address, Memory_Address_4G, bool, FALSE](addresses′))
{-7} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(f′(cons1_var′))),
extend[Address, Memory_Address_4G, bool, FALSE](addresses′))

{1} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(reduce(ok_result(null),
λ
(e:
[Memory_Address_4G,
list[Byte]],
r:
[Linear_memory
[Physical_memory, pm_phy] →
ExprResult
[Linear_memory
[Physical_memory, pm_phy]
list[Byte]]]):
(r
##
(λ
(tail: list[Byte]):
(f′(e)
##
(λ
(head: list[Byte]):
ok_result
(append
(head, tail)))))))
(cons2_var′))),
extend[Address, Memory_Address_4G, bool, FALSE](addresses′))
{2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
(reduce(ok_result(null),
λ
(e:
[Memory_Address_4G,
list[Byte]],
r:
[Linear_memory
[Physical_memory,
pm_phy] →
ExprResult
[Linear_memory
[Physical_memory,
pm_phy],
list[Byte]]]):

Hiding formulas: 2,

Expanding the definition of apply_side_effects,

Expanding the definition of expr_2_super,

which is trivially true.

This completes the proof of `apply_side_effects_unchanged.2.3`.

apply_side_effects_unchanged.2.4:

```
{-1}  plain_memory?(pm_phy)
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states = pm_phy`states
{-4}  (addresses' ⊆ restrict [Address, Memory_Address_4G, boolean]((pm_phy`ro_addr ∪ pm_phy`rw_addr)))
{-5}  every(λ (e: [Memory_Address_4G, list[Byte]]):
            unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                        singleton(expr_2_super(f'(e))),
                                        extend[Address, Memory_Address_4G, bool, FALSE]
                                              (addresses')))
          (cons2_var')
{-6}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(apply_side_effects(cons2_var', f'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
{-7}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(f'(cons1_var'))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
-------------------------------------------------------------------------------------------------
{1}   (extend[Address, Memory_Address_4G, bool, FALSE](addresses') ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(λ (s:
                                                           Linear_memory
                                                           [Physical_memory, pm_phy]):
                                                         (reduce(ok_result(null),
                                                                 λ
                                                                 (e:
                                                                    [Memory_Address_4G,
                                                                      list[Byte]],
                                                                  r:
                                                                    [Linear_memory
                                                                       [Physical_memory,
                                                                         pm_phy] →
                                                                       Ex-
                                                                       prResult
                                                                       [Linear_memory
                                                                          [Physical_memory,
                                                                            pm_phy],
                                                                          list[Byte]]]):
                                                                 (r
                                                                    ##
                                                                    (λ
                                                                       (tail: list[Byte]):
                                                                       (f'(e)
                                                                          ##
                                                                          (λ
                                                                             (head: list[Byte]):
                                                                             ok_result
                                                                             (append
                                                                                (head,
                                                                                   tail)))))))
                                                                 (cons2_var')
                                                               ##
                                                               (λ
                                                                  (tail_1: list[Byte]):
                                                                  (f'(cons1_var')
                                                                     ##   1625
                                                                     (λ
                                                                        (head_1: list[Byte]):
                                                                        ok_result
                                                                        (append
                                                                           (head_1, tail_1))))))
                                                               (s))),
                                  extend[Address, Memory_Address_4G, bool, FALSE](addresses'))
```

Keeping (-4 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `apply_side_effects_unchanged.2.4`.
Q.E.D.

C.117.28 Linear_Memory_Properties.subset_ptab_address_rw_addr

Terse proof for `subset_ptab_address_rw_addr`.
`subset_ptab_address_rw_addr:`

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], base: Memory_Address_4G):
(address_in_pt_range?(s, (restrict [Address, Memory_Address_4G, bool](pm‘ro_addr) ∪ restrict
∧
(address_block(base, expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, bool, FALSE
⊃ (address_block(base, expt(2, pe_size)) ⊆ pm_phy‘rw_addr)

Repeatedly Skolemizing and flattening,
Case splitting on union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)),
we get 2 subgoals:
`subset_ptab_address_rw_addr.1:`

{-1} (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Ad
= restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} Mem?(base′‘type_of)
{-3} 0 ≤ base′‘offset
{-4} base′‘offset < max_linear_offset
{-5} (address_in_pt_range?(s′, (restrict [Address, Memory_Address_4G, bool](pm′‘ro_addr) ∪ restrict
{-6} (address_block(base′, expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, bool, FALSE](
{1} (address_block(base′, expt(2, pe_size)) ⊆ pm_phy‘rw_addr)

Replacing using formula -1,
Using lemma subset_transitive,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-6 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -2 with the terms: (x!1),
we get 2 subgoals:
`subset_ptab_address_rw_addr.1.1:`

{-1} (x′ ∈ extend [Address, Memory_Address_4G, bool, FALSE](address_in_pt_range?(s′, restrict [Ad
{-2} (x′ ∈ address_in_pt_range?(s′, restrict [Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ p
⇒ (x′ ∈ restrict[Address, Memory_Address_4G, bool](pm_phy‘rw_addr))
{1} (x′ ∈ pm_phy‘rw_addr)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_ptab_address_rw_addr.1.1`.

`subset_ptab_address_rw_addr.1.2:`

{-1} (x' ∈ extend [Address, Memory_Address_4G, bool, FALSE](address_in_pt_range?(s', restrict [Address, Mem
{1} Mem?(x'`type_of) ∧ 0 ≤ $x'$`offset ∧ x'`offset < max_linear_offset
{2} (x' ∈ pm_phy`rw_addr)

Expanding the definition of member,
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `subset_ptab_address_rw_addr.1.2`.
`subset_ptab_address_rw_addr.2:`

{-1} Mem?(base′`type_of)
{-2} 0 ≤ base′`offset
{-3} base′`offset < max_linear_offset
{-4} (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, bool](pm′`ro_addr) ∪ restrict [Address,
{-5} (address_block(base′, expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, bool, FALSE](address_in_
{1} (restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Address, Memory_Address_4G, b
= restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr))
{2} (address_block(base′, expt(2, pe_size)) ⊆ pm_phy`rw_addr)

Keeping (1) and hiding *,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_ptab_address_rw_addr.2`.
Q.E.D.

C.117.29 Linear_Memory_Properties.translate_transformer_invariant

Terse proof for `translate_transformer_invariant`.
`translate_transformer_invariant:`

{1} ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], base: PTab_Address, lin_a: Memory_Address_4G,
access: Memory_access, priv: Memory_privilege):
union(pm`ro_addr, pm`rw_addr)(lin_a) ∧
is_linear_plain_memory?(pm) ∧
(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
⊃
transformer_invariant?(pm`states,
singleton(expr_2_super(translate(lvl,
base,
lin_a,
access,
priv))))

Repeatedly Skolemizing and flattening,
Expanding the definition of translate,
Using lemma pm_states,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Rewriting using fexpr_composition_transformer_invariant [Physical_memory, (range_pt(lvl!1)), [bool, Address]], matching in * where P gets LAMBDA (p: (range_pt(lvl!1))): TRUE,
we get 3 subgoals:

translate_transformer_invariant.1:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states = pm_phy‘states
{-3}  lvl′ < max_level
{-4}  Mem?(base′‘type_of)
{-5}  0 ≤ base′‘offset
{-6}  base′‘offset < max_linear_offset
{-7}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-8}  Mem?(lin_a′‘type_of)
{-9}  0 ≤ lin_a′‘offset
{-10} lin_a′‘offset < max_linear_offset
{-11} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-12} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
------------------------------------------------------------------------------
{1}   transformer_invariant?(pm_phy‘states,
                             singleton(expr_2_super(read_data(pm_phy, pag-
      ing_data_type(lvl′))
                                                    (xlat_idx
                                                      (lvl′, base′, lin_a′)))))
{2}   transformer_invariant?(pm_phy‘states,
                             singleton(expr_2_super((read_data(pm_phy, pag-
      ing_data_type(lvl′))
                                                      (xlat_idx(lvl′, base′, lin_
                                                  ##
                                                  (λ (pe: (range_pt(lvl′))):
                                                      IF pag-
      ing_type?(lvl′, pe)
                                                        THEN IF present?(pe)
                                                             THEN IF ac-
      cessible?(pe, access′)
                                                                     ∧
                                                                     privi-
      leged?(pe, priv′)
                                                                  THEN write_data
                                                                       (pm_phy,
                                                                         pag-
      ing_data_type
                                                                         (lvl′))
                                                                       (xlat_idx
                                                                         (lvl′,
                                                                           base′,
                                                                           lin_a′)
                                                                       set_refer
                                                                         (pe, ac
                                                                       ##
                                                                       ok_result
                                                                       (is_leaf?(p
                                                                         base(pe)
                                                                  ELSE raise_fault
                                                                       (priv′,
                                                                         access′,
                                                                         TRUE,
                                                                         lin_a′)
                                                                  ENDIF
                                                        ELSE raise_fault
                                                             (priv′,
                                                               access′,
                                                               FALSE,
                                                               lin_a′)
                                                        ENDIF
                                                  ELSE fatal_result
                                                  ENDIF)))))
```

Hiding formulas: 2,

Using lemma pm_plain_phy,

Rewriting using plain_memory_transformer_invariant_read_data, matching in *,

Expanding the definition of in_blessed_memory?,

Rewriting using paging_data_type_size, matching in *,

Keeping (-13 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(xlat_idx(lvl!1, base!1, lin_a!1), expt(2, pe_size)), b gets pm_phy`rw_addr,

Rewriting using union_subset3, matching in *,

This completes the proof of `translate_transformer_invariant.1`.

translate_transformer_invariant.2:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states = pm_phy‘states
{-3}  lvl′ < max_level
{-4}  Mem?(base′‘type_of)
{-5}  0 ≤ base′‘offset
{-6}  base′‘offset < max_linear_offset
{-7}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-8}  Mem?(lin_a′‘type_of)
{-9}  0 ≤ lin_a′‘offset
{-10} lin_a′‘offset < max_linear_offset
{-11} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-12} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
------------------------------------------------------------------------------
{1}   ∀ (d: (λ (p: (range_pt(lvl′))): TRUE)):
        transformer_invariant?(pm_phy‘states,
                               IF paging_type?(lvl′, d)
                                 THEN IF present?(d)
                                        THEN IF accessible?(d, access′) ∧ privi-
leged?(d, priv′)
                                               THEN singleton(expr_2_super(write_data
                                                                             (pm_phy,
                                                                               pag-
ing_data_type
                                                                                 (lvl′))
                                                                             (xlat_idx
                                                                               (lvl′,
                                                                                 base′,
                                                                                 lin_a′),
                                                                               set_refere
                                                                               (d, acces
                                                                             ##
                                                                             ok_result
                                                                             (is_leaf?(d),
                                                                               base(d)))
                                                    ELSE singleton(expr_2_super(raise_fault
                                                                                  (priv′,
                                                                                    access′,
                                                                                    TRUE,
                                                                                    lin_a′)))
                                               ENDIF
                                      ELSE singleton(expr_2_super(raise_fault
                                                                    (priv′,
                                                                      access′,
                                                                      FALSE,
                                                                      lin_a′)))
                                 ENDIF
                               ELSE singleton(expr_2_super(fatal_result))
                               ENDIF)
{2}   transformer_invariant?(pm_phy‘states,
                              singleton(expr_2_super((read_data(pm_phy, pag-
ing_data_type(lvl′))
                                                       (xlat_idx(lvl′, base′, lin_
                                                     ##
                                                     (λ (pe: (range_pt(lvl′))):
                                                        IF pag-
ing_type?(lvl′, pe)
                                                          THEN IF present?(pe)
                                                                 THEN IF ac-
cessible?(pe, access′)
                                                                        ∧
                                                                        privi-
leged?(pe, priv′)
                                                                      THEN write_data
                                                                             (pm_phy,
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

`translate_transformer_invariant.2.1:`

```
  {-1}   range_pt(lvl′)(d′)
  {-2}   paging_type?(lvl′, d′)
  {-3}   present?(d′)
  {-4}   accessible?(d′, access′)
  {-5}   privileged?(d′, priv′)
  {-6}   is_linear_plain_memory?(pm′)
  {-7}   pm′`states = pm_phy`states
  {-8}   lvl′ < max_level
  {-9}   Mem?(base′`type_of)
  {-10}  0 ≤ base′`offset
  {-11}  base′`offset < max_linear_offset
  {-12}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
  {-13}  Mem?(lin_a′`type_of)
  {-14}  0 ≤ lin_a′`offset
  {-15}  lin_a′`offset < max_linear_offset
  {-16}  union(pm′`ro_addr, pm′`rw_addr)(lin_a′)
  {-17}  (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
  ------------------------------------------------------------------------------------------
   {1}   transformer_invariant?(pm_phy`states,
                                singleton(expr_2_super(write_data(pm_phy, pag-
         ing_data_type(lvl′))
                                                        (xlat_idx(lvl′, base′, lin_a′),
                                                          set_reference(d′, access′))
                                                    ## ok_result(is_leaf?(d′), base(d′)))))
```

Rewriting using expr_composition_transformer_invariant, matching in *,

we get 2 subgoals:

`translate_transformer_invariant.2.1.1:`

{-1} range_pt(lvl′)(d′)
{-2} paging_type?(lvl′, d′)
{-3} present?(d′)
{-4} accessible?(d′, access′)
{-5} privileged?(d′, priv′)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states = pm_phy‘states
{-8} lvl′ < max_level
{-9} Mem?(base′‘type_of)
{-10} 0 ≤ base′‘offset
{-11} base′‘offset < max_linear_offset
{-12} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-13} Mem?(lin_a′‘type_of)
{-14} 0 ≤ lin_a′‘offset
{-15} lin_a′‘offset < max_linear_offset
{-16} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-17} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)

{1} transformer_invariant?(pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a
set_reference(d′, access′)

{2} transformer_invariant?(pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a
set_reference(d′, access′)
ok_result(is_leaf?(d′), base(d′)))

Hiding formulas: 2,

Using lemma pm_plain_phy,

Rewriting using plain_memory_transformer_invariant_write_data, matching in *,

Expanding the definition of in_blessed_memory?,

Rewriting using paging_data_type_size, matching in *,

This completes the proof of `translate_transformer_invariant.2.1.1`.

`translate_transformer_invariant.2.1.2:`

```
  {-1}  range_pt(lvl′)(d′)
  {-2}  paging_type?(lvl′, d′)
  {-3}  present?(d′)
  {-4}  accessible?(d′, access′)
  {-5}  privileged?(d′, priv′)
  {-6}  is_linear_plain_memory?(pm′)
  {-7}  pm′`states = pm_phy`states
  {-8}  lvl′ < max_level
  {-9}  Mem?(base′`type_of)
 {-10}  0 ≤ base′`offset
 {-11}  base′`offset < max_linear_offset
 {-12}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
 {-13}  Mem?(lin_a′`type_of)
 {-14}  0 ≤ lin_a′`offset
 {-15}  lin_a′`offset < max_linear_offset
 {-16}  union(pm′`ro_addr, pm′`rw_addr)(lin_a′)
 {-17}  (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
-------------------------------------------------------------------------------
  {1}   transformer_invariant?(pm_phy`states,
                               singleton(expr_2_super(ok_result(is_leaf?(d′), base(d′)))))
  {2}   transformer_invariant?(pm_phy`states,
                               singleton(expr_2_super(write_data(pm_phy, pag-
        ing_data_type(lvl′))
                                                      (xlat_idx(lvl′, base′, lin_a′),
                                                       set_reference(d′, access′))
                                                    ## ok_result(is_leaf?(d′), base(d′)))))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `translate_transformer_invariant.2.1.2`.
`translate_transformer_invariant.2.2:`

```
  {-1}  range_pt(lvl′)(d′)
  {-2}  paging_type?(lvl′, d′)
  {-3}  present?(d′)
  {-4}  is_linear_plain_memory?(pm′)
  {-5}  pm′`states = pm_phy`states
  {-6}  lvl′ < max_level
  {-7}  Mem?(base′`type_of)
  {-8}  0 ≤ base′`offset
  {-9}  base′`offset < max_linear_offset
 {-10}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
 {-11}  Mem?(lin_a′`type_of)
 {-12}  0 ≤ lin_a′`offset
 {-13}  lin_a′`offset < max_linear_offset
 {-14}  union(pm′`ro_addr, pm′`rw_addr)(lin_a′)
 {-15}  (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
-------------------------------------------------------------------------------
  {1}   privileged?(d′, priv′)
  {2}   transformer_invariant?(pm_phy`states,
                               singleton(expr_2_super(raise_fault(priv′,
                                                                  access′,
                                                                  TRUE,
                                                                  lin_a′))))
```

Rewriting using raise_fault_transformer_invariant, matching in * where pm gets pm!1,
This completes the proof of `translate_transformer_invariant.2.2`.
`translate_transformer_invariant.2.3`:

```
{-1}  range_pt(lvl′)(d′)
{-2}  paging_type?(lvl′, d′)
{-3}  present?(d′)
{-4}  is_linear_plain_memory?(pm′)
{-5}  pm′`states = pm_phy`states
{-6}  lvl′ < max_level
{-7}  Mem?(base′`type_of)
{-8}  0 ≤ base′`offset
{-9}  base′`offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-11} Mem?(lin_a′`type_of)
{-12} 0 ≤ lin_a′`offset
{-13} lin_a′`offset < max_linear_offset
{-14} union(pm′`ro_addr, pm′`rw_addr)(lin_a′)
{-15} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
------------------------------------------------------------------------------
 {1}  accessible?(d′, access′)
 {2}  transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(raise_fault(priv′,
                                                                access′,
                                                                TRUE,
                                                                lin_a′))))
```

Rewriting using raise_fault_transformer_invariant, matching in * where pm gets pm!1,
This completes the proof of `translate_transformer_invariant.2.3`.
`translate_transformer_invariant.2.4`:

```
{-1}  range_pt(lvl′)(d′)
{-2}  paging_type?(lvl′, d′)
{-3}  is_linear_plain_memory?(pm′)
{-4}  pm′`states = pm_phy`states
{-5}  lvl′ < max_level
{-6}  Mem?(base′`type_of)
{-7}  0 ≤ base′`offset
{-8}  base′`offset < max_linear_offset
{-9}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-10} Mem?(lin_a′`type_of)
{-11} 0 ≤ lin_a′`offset
{-12} lin_a′`offset < max_linear_offset
{-13} union(pm′`ro_addr, pm′`rw_addr)(lin_a′)
{-14} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
------------------------------------------------------------------------------
 {1}  present?(d′)
 {2}  transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super(raise_fault(priv′,
                                                                access′,
                                                                FALSE,
                                                                lin_a′))))
```

Rewriting using raise_fault_transformer_invariant, matching in * where pm gets pm!1,
This completes the proof of `translate_transformer_invariant.2.4`.

`translate_transformer_invariant.2.5:`

{-1}	range_pt(lvl′)(d′)
{-2}	is_linear_plain_memory?(pm′)
{-3}	pm′‘states = pm_phy‘states
{-4}	lvl′ < max_level
{-5}	Mem?(base′‘type_of)
{-6}	0 ≤ base′‘offset
{-7}	base′‘offset < max_linear_offset
{-8}	aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-9}	Mem?(lin_a′‘type_of)
{-10}	0 ≤ lin_a′‘offset
{-11}	lin_a′‘offset < max_linear_offset
{-12}	union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-13}	(address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{1}	paging_type?(lvl′, d′)
{2}	transformer_invariant?(pm_phy‘states, singleton(expr_2_super(fatal_result)))

Keeping (2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_transformer_invariant.2.5`.

translate_transformer_invariant.3:

```
{-1}  is_linear_plain_memory?(pm')
{-2}  pm'`states = pm_phy`states
{-3}  lvl' < max_level
{-4}  Mem?(base'`type_of)
{-5}  0 ≤ base'`offset
{-6}  base'`offset < max_linear_offset
{-7}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-8}  Mem?(lin_a'`type_of)
{-9}  0 ≤ lin_a'`offset
{-10} lin_a'`offset < max_linear_offset
{-11} union(pm'`ro_addr, pm'`rw_addr)(lin_a')
{-12} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
-------------------------------------------------------------------------------
{1}   ∀ (pe: (range_pt(lvl'))):
        accessible?(pe, access') ∧
         privileged?(pe, priv') ∧ present?(pe) ∧ paging_type?(lvl', pe)
         ⊃ range_pt(lvl')(set_reference(pe, access'))
{2}   transformer_invariant?(pm_phy`states,
                             singleton(expr_2_super((read_data(pm_phy, pag-
      ing_data_type(lvl'))
                                                    (xlat_idx(lvl', base', lin_
                                                ##
                                                (λ (pe: (range_pt(lvl'))):
                                                    IF pag-
      ing_type?(lvl', pe)
                                                      THEN IF present?(pe)
                                                           THEN IF ac-
      cessible?(pe, access')
                                                                    ∧
                                                                    privi-
      leged?(pe, priv')
                                                                THEN write_data
                                                                     (pm_phy,
                                                                       pag-
      ing_data_type
                                                                       (lvl'))
                                                                     (xlat_idx
                                                                       (lvl',
                                                                         base',
                                                                         lin_a')
                                                                       set_refer
                                                                       (pe, ac
                                                                     ##
                                                                     ok_result
                                                                     (is_leaf?(p
                                                                       base(pe)
                                                                ELSE raise_fault
                                                                     (priv',
                                                                       access',
                                                                       TRUE,
                                                                       lin_a')
                                                                ENDIF
                                                           ELSE raise_fault
                                                                (priv',
                                                                  access',
                                                                  FALSE,
                                                                  lin_a')
                                                           ENDIF
                                                    ELSE fatal_result
                                                    ENDIF)))))
```

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `translate_transformer_invariant.3`.
Q.E.D.

C.117.30 Linear_Memory_Properties.translate_unchanged_pm_phy_except_pe_TCC1

Terse proof for `translate_unchanged_pm_phy_except_pe_TCC1`.
`translate_unchanged_pm_phy_except_pe_TCC1`:

{1} ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G):
(∀ (*s*: (pm‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(*s*)))
∧
(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)) ⊆ pm_phy‘rw_addr) ∧
is_linear_plain_memory?(pm) ∧ union(pm‘ro_addr, pm‘rw_addr)(lin_a)
⊃
(∀ (s_1, s_2: (pm‘states)):
OK?[Physical_memory, ((range_pt(lvl)))]
(read_data[Physical_memory, ((range_pt(lvl)))]
(pm_phy, paging_data_type(lvl))
(xlat_idx[Physical_memory, pm_phy](lvl, base, lin_a))(s_1)))

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `translate_unchanged_pm_phy_except_pe_TCC1`.
Q.E.D.

C.117.31 Linear_Memory_Properties.translate_unchanged_pm_phy_except_pe_TCC2

Terse proof for `translate_unchanged_pm_phy_except_pe_TCC2`.
`translate_unchanged_pm_phy_except_pe_TCC2`:

{1} ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G):
(∀ (*s*: (pm‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(*s*)))
∧
(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)) ⊆ pm_phy‘rw_addr) ∧
is_linear_plain_memory?(pm) ∧ union(pm‘ro_addr, pm‘rw_addr)(lin_a)
⊃
(∀ (s_1, s_2: (pm‘states)):
OK?[Physical_memory, ((range_pt(lvl)))]
(read_data[Physical_memory, ((range_pt(lvl)))]
(pm_phy, paging_data_type(lvl))
(xlat_idx[Physical_memory, pm_phy](lvl, base, lin_a))(s_2)))

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

This completes the proof of `translate_unchanged_pm_phy_except_pe_TCC2`.

Q.E.D.

C.117.32 Linear_Memory_Properties.translate_unchanged_pm_phy_except_pe

Terse proof for `translate_unchanged_pm_phy_except_pe`.

`translate_unchanged_pm_phy_except_pe`:

{1} ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], base: PTab_Address, lin_a: Memory_Address_4G,
access: Memory_access, priv: Memory_privilege):
(union(pm‘ro_addr, pm‘rw_addr)(lin_a) ∧
is_linear_plain_memory?(pm) ∧
(address_block(xlat_idx(lvl, base, lin_a), expt(2, pe_size)) ⊆ pm_phy‘rw_addr) ∧
(∀ (s: (pm‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))
(s)))
∧
(∀ (s_1, s_2: (pm‘states)):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))
(xlat_idx(lvl, base, lin_a))(s_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl))
(xlat_idx(lvl, base, lin_a))(s_2)),
Write)))
⊃
LET pe_addr = xlat_idx(lvl, base, lin_a) IN
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(translate(lvl,
base,
lin_a,
access,
priv))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(pe_a

Repeatedly Skolemizing and flattening,

Letting addr_set name difference(union(pm_phy‘ro_addr, pm_phy‘rw_addr), address_block(xlat_idx(lvl!1, base!1, lin_a!1), expt(2, pe_size))),

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Case splitting on subset?(addr_set, union(pm_phy‘ro_addr, pm_phy‘rw_addr)),

we get 2 subgoals:

translate_unchanged_pm_phy_except_pe.1:

{-1} (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} is_linear_plain_memory?(pm′)
{-3} plain_memory?(pm_phy)
{-4} pm′‘states = pm_phy‘states
{-5} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size))) = addr_set
{-6} lvl′ < max_level
{-7} Mem?(base′‘type_of)
{-8} 0 ≤ base′‘offset
{-9} base′‘offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-11} Mem?(lin_a′‘type_of)
{-12} 0 ≤ lin_a′‘offset
{-13} lin_a′‘offset < max_linear_offset
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-15} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-16} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-17} ∀ (s1, s2: (pm′‘states)):
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s2)),
Write)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(translate(lvl′,
base′,
lin_a′,
access′,
priv′))),
addr_set)

Expanding the definition of translate,

Rewriting using plain_memory_unchanged_composition [Physical_memory, (range_pt(lvl!1)), [bool, Address]], matching in * where P gets LAMBDA (p: (range_pt(lvl!1))): FORALL (s: (pm_phy‘states)): OK?(read_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1))(s)) IMPLIES set_reference(data(read_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1)) (s)), Write) = set_reference(p, Write), pm gets pm_phy, addresses gets addr_set,

we get 4 subgoals:

`translate_unchanged_pm_phy_except_pe.1.1:`

```
{-1}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′‘states = pm_phy‘states
{-5}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
        = addr_set
{-6}  lvl′ < max_level
{-7}  Mem?(base′‘type_of)
{-8}  0 ≤ base′‘offset
{-9}  base′‘offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-11} Mem?(lin_a′‘type_of)
{-12} 0 ≤ lin_a′‘offset
{-13} lin_a′‘offset < max_linear_offset
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-15} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-16} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-17} ∀ (s_1, s_2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s_1)),
                      Write)
        =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s_2)),
                      Write)
-------------------------------------------------------------------------------
{1}   ∀ (s_1: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))
                     (s_1))
        ⊃
        (∀ (s: (pm_phy‘states)):
           OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))
                        (s))
           ⊃
           set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                       (xlat_idx(lvl′, base′, lin_a′))(s)),
                         Write)
           =
           set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                       (xlat_idx(lvl′, base′, lin_a′))(s_1)),
                         Write))
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super((read_data(pm_phy,
                                                                    pag-
ing_data_type(lvl′))
                                                           (xlat_idx
                                                              (lvl′, base′, lin_
                                                         ##
                                                         (λ (pe: (range_pt(lvl′))):
                                                            IF pag-
ing_type?(lvl′, pe)
                                                            THEN IF present?(pe)
                                                                 THEN IF ac-
cessible?
                                                                         (pe, acc
                                                                         ∧
                                                                         priv-
ileged?(pe, priv′)
                                                                      THEN write_
                                                                           (pm_ph
                                                                              pag-
ing_data_type
                                                                              (lvl′
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -21 with the terms: (s!1 s!2),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.1`.

`translate_unchanged_pm_phy_except_pe.1.2:`

```
{-1}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′‘states = pm_phy‘states
{-5}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
        = addr_set
{-6}  lvl′ < max_level
{-7}  Mem?(base′‘type_of)
{-8}  0 ≤ base′‘offset
{-9}  base′‘offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-11} Mem?(lin_a′‘type_of)
{-12} 0 ≤ lin_a′‘offset
{-13} lin_a′‘offset < max_linear_offset
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-15} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-16} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-17} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
        =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s2)),
                      Write)
-------------------------------------------------------------------------------
{1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                 singleton(expr_2_super(read_data(pm_phy, pag-
     ing_data_type(lvl′))
                                                        (xlat_idx
                                                          (lvl′, base′, lin_a′)
                                 addr_set)
{2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                 singleton(expr_2_super((read_data(pm_phy,
                                                                   pag-
     ing_data_type(lvl′))
                                                          (xlat_idx
                                                            (lvl′, base′, lin_a
                                                        ##
                                                        (λ (pe: (range_pt(lvl′))):
                                                           IF pag-
     ing_type?(lvl′, pe)
                                                           THEN IF present?(pe)
                                                                THEN IF ac-
     cessible?
                                                                        (pe, acc
                                                                        ∧
                                                                        priv-
     ileged?(pe, priv′)
                                                                     THEN write_
                                                                          (pm_ph
                                                                            pag-
     ing_data_type
                                                                            (lvl′)
                                                                          (xlat_id
                                                                            (lvl′
                                                                              ba
                                                                              lin
                                                                            set_r
                                                                            (pe,
                                                                          ##
                                                                          ok_resu
                                                                          (is_leaf
```

Hiding formulas: 2,

Using lemma plain_memory_unchanged_memory_invariant_read_data[Physical_memory, (range_pt(lvl′))],

Using lemma unchanged_memory_invariant_mono,

Expanding the definition of in_blessed_memory?,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using paging_data_type_size, matching in *,

Simplifying, rewriting, and recording with decision procedures,

Keeping (-15 3) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(xlat_idx(lvl!1, base!1, lin_a!1), expt(2, pe_size)), b gets pm_phy‘rw_addr, c gets union(pm_phy‘ro_addr, pm_phy‘rw_addr),

Rewriting using union_subset3, matching in *,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.2`.

translate_unchanged_pm_phy_except_pe.1.3:

```
{-1}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2}  is_linear_plain_memory?(pm′)
{-3}  plain_memory?(pm_phy)
{-4}  pm′‘states = pm_phy‘states
{-5}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
        = addr_set
{-6}  lvl′ < max_level
{-7}  Mem?(base′‘type_of)
{-8}  0 ≤ base′‘offset
{-9}  base′‘offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-11} Mem?(lin_a′‘type_of)
{-12} 0 ≤ lin_a′‘offset
{-13} lin_a′‘offset < max_linear_offset
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-15} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-16} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-17} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
        =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s2)),
                      Write)
-------------------------------------------------------------------------------
{1}  ∀ (d:
        (λ (p: (range_pt(lvl′))):
           ∀ (s: (pm_phy‘states)):
             OK?(read_data(pm_phy, paging_data_type(lvl′))
                          (xlat_idx(lvl′, base′, lin_a′))(s))
              ⊃
             set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                         (xlat_idx(lvl′, base′, lin_a′))(s)),
                           Write)
               = set_reference(p, Write))):
      unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  IF paging_type?(lvl′, d)
                                    THEN IF present?(d)
                                           THEN IF accessible?(d, access′) ∧
                                                   privileged?(d, priv′)
                                                  THEN singleton(expr_2_super
                                                                   (write_data
                                                                     (pm_phy,
                                                                       pag-
ing_data_type(lvl′))
                                                                     (xlat_idx
                                                                       (lvl′, base′, lin_
                                                                       set_reference(d, 
                                                                     ##
                                                                     ok_result
                                                                     (is_leaf?(d), base(d
                                                ELSE singleton(expr_2_super
                                                                 (raise_fault
                                                                   (priv′,
                                                                     access′,
                                                                     TRUE,
                                                                     lin_a′)))
                                           ENDIF
                                    ELSE singleton(expr_2_super(raise_fault
                                                                  (priv′,
                                                                    access′,
                                                                    FALSE,
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

`translate_unchanged_pm_phy_except_pe.1.3.1:`

```
{-1}  range_pt(lvl′)(d′)
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
          = set_reference(d′, Write)
{-3}  paging_type?(lvl′, d′)
{-4}  present?(d′)
{-5}  accessible?(d′, access′)
{-6}  privileged?(d′, priv′)
{-7}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-8}  is_linear_plain_memory?(pm′)
{-9}  plain_memory?(pm_phy)
{-10} pm′‘states = pm_phy‘states
{-11} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
       = addr_set
{-12} lvl′ < max_level
{-13} Mem?(base′‘type_of)
{-14} 0 ≤ base′‘offset
{-15} base′‘offset < max_linear_offset
{-16} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-17} Mem?(lin_a′‘type_of)
{-18} 0 ≤ lin_a′‘offset
{-19} lin_a′‘offset < max_linear_offset
{-20} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-21} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-22} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-23} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
        =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s2)),
                      Write)
-------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(write_data(pm_phy,
                                                                    pag-
ing_data_type(lvl′))
                                                                   (xlat_idx
                                                                     (lvl′, base′, lin_a′
                                                                     set_reference(d′, a
                                                          ##
                                                          ok_result(is_leaf?(d′),
                                                                    base(d′)))),
                                  addr_set)
```

Using lemma plain_memory_unchanged_composition[Physical_memory, Unit, [bool, Address]],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`translate_unchanged_pm_phy_except_pe.1.3.1.1:`

```
{-1}  plain_memory?(pm_phy)
{-2}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-3}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(write_data(pm_phy,
                                                                    pag-
      ing_data_type(lvl′))
                                                         (xlat_idx
                                                           (lvl′, base′, lin_a′
                                                           set_reference(d′, a
                                                  ##
                                                  (λ (u: Unit):
                                                     ok_result
                                                       [Physical_memory,
                                                         [bool, Ad-
      dress]]
                                                       (is_leaf?(d′),
                                                         base(d′)))))),
                                  addr_set)
{-4}  range_pt(lvl′)(d′)
{-5}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
           = set_reference(d′, Write)
{-6}  paging_type?(lvl′, d′)
{-7}  present?(d′)
{-8}  accessible?(d′, access′)
{-9}  privileged?(d′, priv′)
{-10} is_linear_plain_memory?(pm′)
{-11} pm′‘states = pm_phy‘states
{-12} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
       = addr_set
{-13} lvl′ < max_level
{-14} Mem?(base′‘type_of)
{-15} 0 ≤ base′‘offset
{-16} base′‘offset < max_linear_offset
{-17} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-18} Mem?(lin_a′‘type_of)
{-19} 0 ≤ lin_a′‘offset
{-20} lin_a′‘offset < max_linear_offset
{-21} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-22} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-23} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-24} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
         =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s2)),
                       Write)
```

```
{1}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(write_data(pm_phy,
                                                                    pag-
      ing_data_type(lvl′))
                                                         (xlat_idx
                                                           (lvl′, base′, lin_a′
                                                           set_reference(d′, a
                                                  ##
                                                  ok_result(is_leaf?(d′),
```

Expanding the definition of ##,

Expanding the definition of ##,

which is trivially true.

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.1.1`.

`translate_unchanged_pm_phy_except_pe.1.3.1.2:`

```
{-1}  plain_memory?(pm_phy)
{-2}  (addr_set ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-3}  range_pt(lvl′)(d′)
{-4}  ∀ (s: (pm_phy`states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
          = set_reference(d′, Write)
{-5}  paging_type?(lvl′, d′)
{-6}  present?(d′)
{-7}  accessible?(d′, access′)
{-8}  privileged?(d′, priv′)
{-9}  is_linear_plain_memory?(pm′)
{-10} pm′`states = pm_phy`states
{-11} ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
       = addr_set
{-12} lvl′ < max_level
{-13} Mem?(base′`type_of)
{-14} 0 ≤ base′`offset
{-15} base′`offset < max_linear_offset
{-16} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-17} Mem?(lin_a′`type_of)
{-18} 0 ≤ lin_a′`offset
{-19} lin_a′`offset < max_linear_offset
{-20} union(pm′`ro_addr, pm′`rw_addr)(lin_a′)
{-21} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
{-22} ∀ (s: (pm′`states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-23} ∀ (s1, s2: (pm′`states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
         =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s2)),
                       Write)
-------------------------------------------------------------------------------
{1}   ∀ (d: (λ (u: Unit): TRUE)):
        unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                    singleton(expr_2_super(ok_result
                                                           [Physical_memory, [bool
      dress]]
                                                           (is_leaf?(d′), base(d′)))),
                                    addr_set)
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(write_data(pm_phy,
                                                                    pag-
      ing_data_type(lvl′))
                                                                   (xlat_idx
                                                                     (lvl′, base′, lin_a′
                                                                     set_reference(d′, a
                                                         ##
                                                         ok_result(is_leaf?(d′),
                                                                   base(d′)))),
                                  addr_set)
```

Hiding formulas: 2,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.1.2`.

`translate_unchanged_pm_phy_except_pe.1.3.1.3:`

{-1} plain_memory?(pm_phy)
{-2} (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-3} range_pt(lvl′)(d′)
{-4} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
⊃
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s)),
Write)
= set_reference(d′, Write)
{-5} paging_type?(lvl′, d′)
{-6} present?(d′)
{-7} accessible?(d′, access′)
{-8} privileged?(d′, priv′)
{-9} is_linear_plain_memory?(pm′)
{-10} pm′‘states = pm_phy‘states
{-11} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
= addr_set
{-12} lvl′ < max_level
{-13} Mem?(base′‘type_of)
{-14} 0 ≤ base′‘offset
{-15} base′‘offset < max_linear_offset
{-16} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-17} Mem?(lin_a′‘type_of)
{-18} 0 ≤ lin_a′‘offset
{-19} lin_a′‘offset < max_linear_offset
{-20} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-21} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-22} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-23} ∀ (s1, s2: (pm′‘states)):
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s2)),
Write)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy,
pag-
ing_data_type(lvl′))
(xlat_idx
(lvl′, base′, lin_a′
set_reference
(d′, access′)))),
addr_set)
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(write_data(pm_phy,
pag-
ing_data_type(lvl′))
(xlat_idx
(lvl′, base′, lin_a′
set_reference(d′, a
##
ok_result(is_leaf?(d′),
base(d′)))),
addr_set)

Hiding formulas: 2,
Replacing using formula -10,
Using lemma plain_memory_unchanged_memory_invariant_write_data,
Expanding the definition of in_blessed_memory?,
Rewriting using paging_data_type_size, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.1.3`.
`translate_unchanged_pm_phy_except_pe.1.3.2`:

```
{-1}  range_pt(lvl′)(d′)
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
           = set_reference(d′, Write)
{-3}  paging_type?(lvl′, d′)
{-4}  present?(d′)
{-5}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6}  is_linear_plain_memory?(pm′)
{-7}  plain_memory?(pm_phy)
{-8}  pm′‘states = pm_phy‘states
{-9}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)))
        = addr_set
{-10} lvl′ < max_level
{-11} Mem?(base′‘type_of)
{-12} 0 ≤ base′‘offset
{-13} base′‘offset < max_linear_offset
{-14} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-15} Mem?(lin_a′‘type_of)
{-16} 0 ≤ lin_a′‘offset
{-17} lin_a′‘offset < max_linear_offset
{-18} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-19} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-20} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-21} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
         =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s2)),
                       Write)
-------------------------------------------------------------------------------
{1}   privileged?(d′, priv′)
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(raise_fault(priv′,
                                                                     access′,
                                                                     TRUE,
                                                                     lin_a′))),
                                  addr_set)
```

Using lemma raise_fault_unchanged_memory_invariant,

Using lemma unchanged_memory_invariant_mono,
Rewriting using subset_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.2`.
`translate_unchanged_pm_phy_except_pe.1.3.3:`

```
{-1}  range_pt(lvl′)(d′)
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
          = set_reference(d′, Write)
{-3}  paging_type?(lvl′, d′)
{-4}  present?(d′)
{-5}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-6}  is_linear_plain_memory?(pm′)
{-7}  plain_memory?(pm_phy)
{-8}  pm′‘states = pm_phy‘states
{-9}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
       = addr_set
{-10} lvl′ < max_level
{-11} Mem?(base′‘type_of)
{-12} 0 ≤ base′‘offset
{-13} base′‘offset < max_linear_offset
{-14} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-15} Mem?(lin_a′‘type_of)
{-16} 0 ≤ lin_a′‘offset
{-17} lin_a′‘offset < max_linear_offset
{-18} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-19} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-20} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-21} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
         =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s2)),
                       Write)
-------------------------------------------------------------------------------
{1}   accessible?(d′, access′)
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(raise_fault(priv′,
                                                                     access′,
                                                                     TRUE,
                                                                     lin_a′))),
                                  addr_set)
```

Using lemma raise_fault_unchanged_memory_invariant,
Using lemma unchanged_memory_invariant_mono,
Rewriting using subset_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.3`.

`translate_unchanged_pm_phy_except_pe.1.3.4`:

```
{-1}  range_pt(lvl′)(d′)
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
           = set_reference(d′, Write)
{-3}  paging_type?(lvl′, d′)
{-4}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-5}  is_linear_plain_memory?(pm′)
{-6}  plain_memory?(pm_phy)
{-7}  pm′‘states = pm_phy‘states
{-8}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)))
       = addr_set
{-9}  lvl′ < max_level
{-10} Mem?(base′‘type_of)
{-11} 0 ≤ base′‘offset
{-12} base′‘offset < max_linear_offset
{-13} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-14} Mem?(lin_a′‘type_of)
{-15} 0 ≤ lin_a′‘offset
{-16} lin_a′‘offset < max_linear_offset
{-17} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-18} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-19} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-20} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                    (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
         =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s2)),
                       Write)
-------------------------------------------------------------------------------
{1}   present?(d′)
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(raise_fault(priv′,
                                                                     access′,
                                                                     FALSE,
                                                                     lin_a′))),
                                  addr_set)
```

Using lemma raise_fault_unchanged_memory_invariant,

Using lemma unchanged_memory_invariant_mono,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.4`.

`translate_unchanged_pm_phy_except_pe.1.3.5:`

```
{-1}  range_pt(lvl′)(d′)
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
         ⊃
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s)),
                       Write)
          = set_reference(d′, Write)
{-3}  (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-4}  is_linear_plain_memory?(pm′)
{-5}  plain_memory?(pm_phy)
{-6}  pm′‘states = pm_phy‘states
{-7}  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_siz
       = addr_set
{-8}  lvl′ < max_level
{-9}  Mem?(base′‘type_of)
{-10} 0 ≤ base′‘offset
{-11} base′‘offset < max_linear_offset
{-12} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-13} Mem?(lin_a′‘type_of)
{-14} 0 ≤ lin_a′‘offset
{-15} lin_a′‘offset < max_linear_offset
{-16} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-17} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-18} ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-19} ∀ (s1, s2: (pm′‘states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                     (xlat_idx(lvl′, base′, lin_a′))(s1)),
                      Write)
         =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, lin_a′))(s2)),
                       Write)
-------------------------------------------------------------------------------
{1}   paging_type?(lvl′, d′)
{2}   unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, single-
      ton(expr_2_super(fatal_result)),
                                  addr_set)
```

Keeping (2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.3.5`.

translate_unchanged_pm_phy_except_pe.1.4:

{-1} (addr_set ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} is_linear_plain_memory?(pm′)
{-3} plain_memory?(pm_phy)
{-4} pm′‘states = pm_phy‘states
{-5} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size))) = addr_set
{-6} lvl′ < max_level
{-7} Mem?(base′‘type_of)
{-8} 0 ≤ base′‘offset
{-9} base′‘offset < max_linear_offset
{-10} aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-11} Mem?(lin_a′‘type_of)
{-12} 0 ≤ lin_a′‘offset
{-13} lin_a′‘offset < max_linear_offset
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-15} (address_block(xlat_idx(lvl′, base′, lin_a′), expt(2, pe_size)) ⊆ pm_phy‘rw_addr)
{-16} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s))
{-17} ∀ (s1, s2: (pm′‘states)):
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s1)), Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s2)), Write)

{1} ∀ (pe: (range_pt(lvl′))):
accessible?(pe, access′) ∧ privileged?(pe, priv′) ∧ present?(pe) ∧ paging_type?(lvl′, pe) ⊃ range_pt(lvl′)(set_reference(pe, access′))
{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(expr_2_super((read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′)) ## (λ (pe: (range_pt(lvl′))): IF paging_type?(lvl′, pe) THEN IF present?(pe) THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′) THEN write_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′), set_reference(pe, access′)) ## ok_result(is_leaf?(pe), base(pe)) ELSE raise_fault

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `translate_unchanged_pm_phy_except_pe.1.4`.

`translate_unchanged_pm_phy_except_pe.2:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  plain_memory?(pm_phy)
{-3}  pm'`states = pm_phy`states
{-4}  ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_siz
        = addr_set
{-5}  lvl' < max_level
{-6}  Mem?(base'`type_of)
{-7}  0 ≤ base'`offset
{-8}  base'`offset < max_linear_offset
{-9}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base'))
{-10} Mem?(lin_a'`type_of)
{-11} 0 ≤ lin_a'`offset
{-12} lin_a'`offset < max_linear_offset
{-13} union(pm'`ro_addr, pm'`rw_addr)(lin_a')
{-14} (address_block(xlat_idx(lvl', base', lin_a'), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
{-15} ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, paging_data_type(lvl'))(xlat_idx(lvl', base', lin_a'))(s))
{-16} ∀ (s1, s2: (pm'`states)):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                      (xlat_idx(lvl', base', lin_a'))(s1)),
                      Write)
        =
        set_reference(data(read_data(pm_phy, paging_data_type(lvl'))
                                      (xlat_idx(lvl', base', lin_a'))(s2)),
                      Write)
-------------------------------------------------------------------------------
{1}   (addr_set ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2}   unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(translate(lvl',
                                                                   base',
                                                                   lin_a',
                                                                   access',
                                                                   priv'))),
                                  addr_set)
```

Replacing using formula -4,

Rewriting using difference_subset, matching in *,

This completes the proof of `translate_unchanged_pm_phy_except_pe.2`.

Q.E.D.

C.117.33 Linear_Memory_Properties.translate_result_ok_TCC1

Terse proof for `translate_result_ok_TCC1`.

`translate_result_ok_TCC1`:

{1} ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s_1, s_2: (pm‘states), base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G): is_linear_plain_memory?(pm) ∧ union(pm‘ro_addr, pm‘rw_addr)(lin_a) ⊃ Mem?(xlat_idx[Physical_memory, pm_phy](lvl, base, lin_a)‘type_of) ∧ 0 ≤ xlat_idx[Physical_memory, pm_phy](lvl, base, lin_a)‘offset ∧ xlat_idx[Physical_memory, pm_phy](lvl, base, lin_a)‘offset < max_linear_offset

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: (Mem! max_linear! <!)
Using lemma xlat_idx_memory_address,
Using lemma xlat_idx_memory_address2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_result_ok_TCC1`.
Q.E.D.

C.117.34 Linear_Memory_Properties.translate_result_ok

Terse proof for `translate_result_ok`.

`translate_result_ok`:

{1} ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s_1, s_2: (pm‘states), base: PTab_Address, lin_a: Memory_Address_4G, access: Memory_access, priv: Memory_privilege): union(pm‘ro_addr, pm‘rw_addr)(lin_a) ∧ is_linear_plain_memory?(pm) ∧ pe_in_pt_range?(s_1, restrict[Address, Memory_Address_4G, boolean]((pm‘ro_addr ∪ pm‘rw_addr)), lvl)(xlat_idx(lvl, base, lin_a)) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s_1)) ⊃ OK?(translate(lvl, base, lin_a, access, priv)(s_2))

Repeatedly Skolemizing and flattening,
Expanding the definition of translate,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Hiding formulas: (-13 -14 -15 -16),
Expanding the definition of ##,
Expanding the definition of raise_fault,
Expanding the definition of exception_result,
Expanding the definition of fatal_result,
Expanding the definition of ##,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -15 with the terms: (s1!1 s2!1 lvl!1 xlat_idx(lvl!1, base!1, lin_a!1)),
Instantiating the top quantifier in -13 with the terms: (s1!1 s2!1),
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -15 with the terms: (s2!1 lvl!1 xlat_idx(lvl!1, base!1, lin_a!1)),
we get 2 subgoals:

translate_result_ok.1:

{-1} $\mathrm{lvl}' < \mathrm{max\_level}$
{-2} $\mathrm{pm}'\text{`states}(s_1')$
{-3} $\mathrm{pm}'\text{`states}(s_2')$
{-4} $\mathrm{Mem?}(\mathrm{base}'\text{`type\_of})$
{-5} $0 \leq \mathrm{base}'\text{`offset}$
{-6} $\mathrm{base}'\text{`offset} < \mathrm{max\_linear\_offset}$
{-7} $\mathrm{aligned?}(\mathrm{bits\_per\_level}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}] + \mathrm{pe\_size})(\mathrm{offset}(\mathrm{base}'))$
{-8} $\mathrm{Mem?}(\mathrm{lin\_a}'\text{`type\_of})$
{-9} $0 \leq \mathrm{lin\_a}'\text{`offset}$
{-10} $\mathrm{lin\_a}'\text{`offset} < \mathrm{max\_linear\_offset}$
{-11} $\mathrm{union}(\mathrm{pm}'\text{`ro\_addr},\ \mathrm{pm}'\text{`rw\_addr})(\mathrm{lin\_a}')$
{-12} $\mathrm{pm}'\text{`mem} = \mathrm{linear\_pm}$
{-13} $\mathrm{pe\_in\_pdir\_range?}(s_1',\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
$=$
$\mathrm{pe\_in\_pdir\_range?}(s_2',\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
{-14} $\mathrm{pe\_in\_ptab\_range?}(s_1',\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
$=$
$\mathrm{pe\_in\_ptab\_range?}(s_2',\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})))$
{-15} $\mathrm{OK?}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_2'))$
{-16} $\mathrm{set\_reference}(\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1')),\ \mathrm{Write})$
$=$
$\mathrm{set\_reference}(\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_2')),\ \mathrm{Write})$
{-17} $\mathrm{plain\_memory?}(\mathrm{pm\_phy})$
{-18} $\mathrm{pm}'\text{`states} = \mathrm{pm\_phy}\text{`states}$
{-19} $(\mathrm{pm}'\text{`other\_actions} \subseteq \mathrm{pm\_phy}\text{`other\_actions})$
{-20} $(\mathrm{linear\_resolve\_register\_transformers} \subseteq \mathrm{pm\_phy}\text{`other\_actions})$
{-21} $\forall\ (a\colon ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr}))):$
$\mathrm{Mem?}(\mathrm{type\_of}(a)) \wedge 0 \leq \mathrm{offset}(a) \wedge \mathrm{offset}(a) < \mathrm{max\_linear\_offset}$
{-22} $\forall\ (s\colon (\mathrm{pm}'\text{`states})):$
$\mathrm{linear\_blessed?}(s,\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}](\mathrm{pm}'\text{`ro\_addr}),\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}](\mathrm{pm}'\text{`rw\_addr}))$
{-23} $\mathrm{pe\_in\_pt\_range?}(s_1',\ \mathrm{restrict}[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})),\ \mathrm{lvl}')(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))$
{-24} $\mathrm{OK?}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1'))$
{-25} $\mathrm{paging\_type?}(\mathrm{lvl}',\ \mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1')))$
{-26} $\mathrm{present?}(\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1')))$
{-27} $\mathrm{accessible?}(\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1')),\ \mathrm{access}')$
{-28} $\mathrm{privileged?}(\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1')),\ \mathrm{priv}')$
{-29} $\mathrm{OK?}((\mathrm{write\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'),\ \mathrm{set\_reference}(\mathrm{data}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{paging\_data\_type}(\mathrm{lvl}'))(\mathrm{xlat\_idx}(\mathrm{lvl}',\ \mathrm{base}',\ \mathrm{lin\_a}'))(s_1')),\ \mathrm{Write}))(s_2'))$

Expanding the definition of ##,

Using lemma set_reference_present,

Using lemma set_reference_base,

Using lemma set_reference_accessible,

Using lemma set_reference_privileged,

Using lemma set_reference_leaf,

Using lemma set_reference_paging_type,

Case splitting on OK?(write_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1), set_reference(data(read_data (pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1)) (s2!1)), access!1)) (state(read_data(pm_phy, paging_data_type(lvl!1)) (xlat_idx(lvl!1, base!1, lin_a!1)) (s2!1)))),

we get 4 subgoals:

`translate_result_ok.1.1:`

{-1} OK?(write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′),
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
access′))
(state(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2'))))

{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
⊃
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
=
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')))

{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')))

{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
priv′)
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
priv′)

{-5} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)

Installing automatic rewrites from: ## ok_result

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_ok.1.1`.

translate_result_ok.1.2:

{-1} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
⊃
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
=
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)))

{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)))

{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
priv′)
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
priv′)

{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
⊃
accessible?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
access′)

Hiding formulas: 2,

Using lemma plain_memory_write_data_ok,

Case splitting on in_blessed_memory?(paging_data_type(lvl!1), xlat_idx(lvl!1, base!1, lin_a!1), pm_phy‘rw_addr),

we get 2 subgoals:

translate_result_ok.1.2.1:

{-1} in_blessed_memory?(paging_data_type(lvl′), xlat_idx(lvl′, base′, lin_a′), pm_phy‘rw_addr)
{-2} plain_memory?(pm_phy) ∧
pm_phy‘states
(state(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))
(s_2')))
∧
in_blessed_memory?(paging_data_type(lvl′), xlat_idx(lvl′, base′, lin_a′),
pm_phy‘rw_addr)
⊃
OK?(write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′),
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
access′))
(state(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2'))))
{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
⊃
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
=
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')))
{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')))
{-5} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
priv′)
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))

Using lemma plain_memory_transformer_invariant_read_data,

Using lemma expr_transformer_invariant_next_ok,

Expanding the definition of singleton,

Installing automatic rewrites from: has_next_state in_blessed_memory_rw_ro

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_ok.1.2.1`.

translate_result_ok.1.2.2:

{-1} plain_memory?(pm_phy) ∧
pm_phy‘states
(state(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))
(s_2')))
∧
in_blessed_memory?(paging_data_type(lvl′), xlat_idx(lvl′, base′, lin_a′),
pm_phy‘rw_addr)
⊃
OK?(write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′),
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
access′))
(state(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2'))))

{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
⊃
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
=
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')))

{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')))

{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_1')),
priv′)
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s_2')),

Expanding the definition of in_blessed_memory?,

Rewriting using paging_data_type_size, matching in *,

Hiding formulas: (-1 -2 -3 -4 -5 -6 -7 2),

Using lemma pe_in_pt_address_in_pt,

Using lemma subset_ptab_address_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -24 with the terms: (s1!1),

Expanding the definition of linear_blessed?,

which is trivially true.

This completes the proof of `translate_result_ok.1.2.2`.

`translate_result_ok.1.3:`

{-1} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)),
Write)
⊃
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)))
=
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)))

{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)))

{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)),
priv′)
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)),
priv′)

{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)))
⊃
accessible?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s'_1)),
access′)

Rewriting using set_reference_range, matching in *,

This completes the proof of `translate_result_ok.1.3`.

translate_result_ok.1.4:

{-1} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
⊃
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
=
paging_type?(lvl′,
data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)))

{-2} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
⊃
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
=
is_leaf?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)))

{-3} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
⊃
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
priv′)
=
privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
priv′)

{-4} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′2)),
Write)
∧
present?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)))
⊃
accessible?(data(read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, lin_a′))(s′1)),
access′)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_result_ok.1.4`.

translate_result_ok.2:

```
{-1}  lvl′ < max_level
{-2}  pm′‘states(s′1)
{-3}  pm′‘states(s′2)
{-4}  Mem?(base′‘type_of)
{-5}  0 ≤ base′‘offset
{-6}  base′‘offset < max_linear_offset
{-7}  aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)(offset(base′))
{-8}  Mem?(lin_a′‘type_of)
{-9}  0 ≤ lin_a′‘offset
{-10} lin_a′‘offset < max_linear_offset
{-11} union(pm′‘ro_addr, pm′‘rw_addr)(lin_a′)
{-12} pm′‘mem = linear_pm
{-13} pe_in_pdir_range?(s′1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm′‘ro_addr ∪ pm′‘rw_addr)))
      =
      pe_in_pdir_range?(s′2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-14} pe_in_ptab_range?(s′1,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm′‘ro_addr ∪ pm′‘rw_addr)))
      =
      pe_in_ptab_range?(s′2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-15} set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                   (xlat_idx(lvl′, base′, lin_a′))(s′1)),
                    Write)
      =
      set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                   (xlat_idx(lvl′, base′, lin_a′))(s′2)),
                    Write)
{-16} plain_memory?(pm_phy)
{-17} pm′‘states = pm_phy‘states
{-18} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-19} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-20} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
        Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-21} ∀ (s: (pm′‘states)):
        linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
                        restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-22} pe_in_pt_range?(s′1,
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                      lvl′)
                     (xlat_idx(lvl′, base′, lin_a′))
{-23} OK?(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))(s′1))
{-24} paging_type?(lvl′,
                   data(read_data(pm_phy, paging_data_type(lvl′))
                                  (xlat_idx(lvl′, base′, lin_a′))(s′1)))
{-25} present?(data(read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, lin_a′))
                            (s′1)))
{-26} accessible?(data(read_data(pm_phy, paging_data_type(lvl′))
                                 (xlat_idx(lvl′, base′, lin_a′))(s′1)),
                  access′)
{-27} privileged?(data(read_data(pm_phy, paging_data_type(lvl′))
                                 (xlat_idx(lvl′, base′, lin_a′))(s′1)),
                  priv′)
{-28} OK?((write_data(pm_phy, paging_data_type(lvl′))
                      (xlat_idx(lvl′, base′, lin_a′),
                        set_reference(data(read_data(pm_phy, paging_data_type(lvl′))
                                                     (xlat_idx(lvl′, base′, lin_a′))(s′1)),
                                      access′))
```

Expanding the definition of pe_in_pt_range?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_result_ok.2`.
Q.E.D.

C.117.35 Linear_Memory_Properties.translate_result_as_read_TCC1

Terse proof for `translate_result_as_read_TCC1`.
`translate_result_as_read_TCC1`:

```
{1}  ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s: Lin-
     ear_memory[Physical_memory, pm_phy],
           base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G,
           access: Memory_access, priv: Memory_privilege):
       union(pm`ro_addr, pm`rw_addr)(lin_a) ∧
        is_linear_plain_memory?(pm) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s))
        ⊃
        (∀ (res: ExprResult[Physical_memory, ((range_pt(lvl)))]):
             res = read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s) ⊃
              OK?[Physical_memory, ((range_pt(lvl)))](res))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of translate,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_result_as_read_TCC1`.
Q.E.D.

C.117.36 Linear_Memory_Properties.translate_result_as_read_TCC2

Terse proof for `translate_result_as_read_TCC2`.
`translate_result_as_read_TCC2`:

```
{1}  ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s: Lin-
     ear_memory[Physical_memory, pm_phy],
           base: PTab_Address[Physical_memory, pm_phy], lin_a: Memory_Address_4G,
           access: Memory_access, priv: Memory_privilege):
       union(pm`ro_addr, pm`rw_addr)(lin_a) ∧
        is_linear_plain_memory?(pm) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s))
        ⊃
        (∀ (res: ExprResult[Physical_memory, ((range_pt(lvl)))]):
             res = read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s) ⊃
              present?(data[Physical_memory, ((range_pt(lvl)))](res)))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -10,
Expanding the definition of translate,
Installing automatic rewrites from: (##! raise_fault! fatal_result! exception_result!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_result_as_read_TCC2`.
Q.E.D.

C.117.37 Linear_Memory_Properties.translate_result_as_read

Terse proof for `translate_result_as_read`.

`translate_result_as_read`:

```
{1}  ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          base: PTab_Address, lin_a: Memory_Address_4G, access: Memory_access, priv: Memory_privilege):
       union(pm`ro_addr, pm`rw_addr)(lin_a) ∧
        is_linear_plain_memory?(pm) ∧ OK?(translate(lvl, base, lin_a, access, priv)(s))
        ⊃
        LET res = read_data(pm_phy, paging_data_type(lvl))(xlat_idx(lvl, base, lin_a))(s) IN
          data(translate(lvl, base, lin_a, access, priv)(s)) =
           (is_leaf?(data(res)), base(data(res)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of translate,
Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res! ok_result! fatal_result! raise_fault! exception_result!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_result_as_read`.
Q.E.D.

C.117.38 Linear_Memory_Properties.translate_same_result

Terse proof for `translate_same_result`.

`translate_same_result`:

```
{1}  ∀ (lvl: Level, pm: Plain_Memory[Linear_memory], s1, s2: (pm`states), base: PTab_Address
          lin_a: Memory_Address_4G, ac1, ac2: Memory_access, priv: Memory_privilege):
       union(pm`ro_addr, pm`rw_addr)(lin_a) ∧
        is_linear_plain_memory?(pm) ∧
         pe_in_pt_range?(s1,
                         restrict[Address, Memory_Address_4G, boolean]
                             ((pm`ro_addr ∪ pm`rw_addr)),
                         lvl)
                        (xlat_idx(lvl, base, lin_a))
          ∧
          OK?(translate(lvl, base, lin_a, ac1, priv)(s1)) ∧
           OK?(translate(lvl, base, lin_a, ac2, priv)(s2))
        ⊃
        data(translate(lvl, base, lin_a, ac1, priv)(s1)) =
         data(translate(lvl, base, lin_a, ac2, priv)(s2))
```

Repeatedly Skolemizing and flattening,
Using lemma translate_result_as_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma translate_result_as_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of is_linear_plain_memory?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -9 with the terms: (s1!1 s2!1 lvl!1 xlat_idx(lvl!1, base!1, lin_a!1)),
Using lemma set_reference_leaf,
Using lemma set_reference_base,
Using lemma set_reference_present,
Keeping (-1 -2 -3 -12 -19 -20 -21 -22 1) and hiding *,
Replacing using formula -6,
Replacing using formula -8,
Hiding formulas: (-6 -8),
Expanding the definition of translate,
Installing automatic rewrites from: (##! expr_2_super! raise_fault! expr_2_super_res! exception_result! ok_result! fatal_result!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_same_result`.
Q.E.D.

C.117.39 Linear_Memory_Properties.xlat_pe_in_pdir_range_TCC1

Terse proof for `xlat_pe_in_pdir_range_TCC1`.
`xlat_pe_in_pdir_range_TCC1`:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G,
s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧ pm‘states(s_1) ∧ pm‘states(s_2)
⊃ pdir_lvl[Physical_memory, pm_phy] < max_level

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_pe_in_pdir_range_TCC1`.
Q.E.D.

C.117.40 Linear_Memory_Properties.xlat_pe_in_pdir_range_TCC2

Terse proof for `xlat_pe_in_pdir_range_TCC2`.
`xlat_pe_in_pdir_range_TCC2`:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G,
s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧ pm‘states(s_1) ∧ pm‘states(s_2)
⊃
OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_2))

Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Instantiating quantified variables,
This completes the proof of `xlat_pe_in_pdir_range_TCC2`.
Q.E.D.

C.117.41 Linear_Memory_Properties.xlat_pe_in_pdir_range_TCC3

Terse proof for `xlat_pe_in_pdir_range_TCC3`.

`xlat_pe_in_pdir_range_TCC3:`

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G,
s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
union(pm`ro_addr, pm`rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧ pm`states($s_1$) ∧ pm`states(s_2)
⊃
Mem?(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2))`base_addr,
addr)`type_of)
∧
0 ≤
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2))`base_addr,
addr)`offset
∧
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2))`base_addr,
addr)`offset
< max_linear_offset

Installing automatic rewrites from: (bits_per_level pe_size Mem max_linear < max_level)

Repeatedly Skolemizing and flattening,

Case splitting on OK?[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s2!1)),

we get 2 subgoals:

xlat_pe_in_pdir_range_TCC3.1:

{-1} OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_2'))
{-2} Mem?(addr′`type_of)
{-3} $0 \leq$ addr′`offset
{-4} addr′`offset $<$ max_linear_offset
{-5} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′`states(s_1')
{-8} pm′`states(s_2')

{1} Mem?(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)`type_of)
∧
$0 \leq$
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)`offset
∧
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)`offset
$<$ max_linear_offset

Using lemma xlat_idx_memory_address,

we get 2 subgoals:

`xlat_pe_in_pdir_range_TCC3.1.1:`

```
{-1}  aligned?(bits_per_level + pe_size)
                (offset
                   (data[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                         (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr))
       ⊃
       xlat_idx(pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                addr')
        < max_linear
{-2}  OK?[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2))
{-3}  Mem?(addr'`type_of)
{-4}  0 ≤ addr'`offset
{-5}  addr'`offset < max_linear_offset
{-6}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-7}  is_linear_plain_memory?(pm')
{-8}  pm'`states(s'_1)
{-9}  pm'`states(s'_2)
------------------------------------------------------------------------
{1}   Mem?(xlat_idx[Physical_memory, pm_phy]
                (pdir_lvl[Physical_memory, pm_phy],
                   data[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                         (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                   addr')`type_of)
       ∧
       0 ≤
        xlat_idx[Physical_memory, pm_phy]
             (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                addr')`offset
        ∧
        xlat_idx[Physical_memory, pm_phy]
             (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                addr')`offset
         < max_linear_offset
```

Using lemma xlat_idx_memory_address2,

we get 2 subgoals:

xlat_pe_in_pdir_range_TCC3.1.1.1:

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr))
⊃
0 ≤
offset
(xlat_idx(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′))

{-2} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr))
⊃
xlat_idx(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)
< max_linear

{-3} OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_2'))

{-4} Mem?(addr′`type_of)

{-5} 0 ≤ addr′`offset

{-6} addr′`offset < max_linear_offset

{-7} union(pm′`ro_addr, pm′`rw_addr)(addr′)

{-8} is_linear_plain_memory?(pm′)

{-9} pm′`states(s_1')

{-10} pm′`states(s_2')

{1} Mem?(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)`type_of)
∧
0 ≤
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)`offset
∧
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))`base_addr,
addr′)`offset
< max_linear_offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_pe_in_pdir_range_TCC3.1.1.1`.
`xlat_pe_in_pdir_range_TCC3.1.1.2`:

```
{-1}  aligned?(bits_per_level + pe_size)
                (offset
                   (data[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                         (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr))
       ⊃
       xlat_idx(pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                addr')
        < max_linear
{-2}  OK?[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_2))
{-3}  Mem?(addr'`type_of)
{-4}  0 ≤ addr'`offset
{-5}  addr'`offset < max_linear_offset
{-6}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-7}  is_linear_plain_memory?(pm')
{-8}  pm'`states(s'_1)
{-9}  pm'`states(s'_2)
------------------------------------------------------------------------
{1}   pdir_lvl[Physical_memory, pm_phy] < 2
{2}   Mem?(xlat_idx[Physical_memory, pm_phy]
                (pdir_lvl[Physical_memory, pm_phy],
                   data[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                         (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                   addr')`type_of)
       ∧
       0 ≤
        xlat_idx[Physical_memory, pm_phy]
             (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                addr')`offset
        ∧
        xlat_idx[Physical_memory, pm_phy]
             (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_2))`base_addr,
                addr')`offset
          < max_linear_offset
```

Using lemma xlat_pe_in_pdir_range_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_pe_in_pdir_range_TCC3.1.1.2`.

`xlat_pe_in_pdir_range_TCC3.1.2:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{OK?}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \quad (\mathrm{read\_data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}](\mathrm{pm\_phy},\ \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s_2')) \\
\{-2\} & \mathrm{Mem?}(\mathrm{addr}'\text{`type\_of}) \\
\{-3\} & 0 \le \mathrm{addr}'\text{`offset} \\
\{-4\} & \mathrm{addr}'\text{`offset} < \mathrm{max\_linear\_offset} \\
\{-5\} & \mathrm{union}(\mathrm{pm}'\text{`ro\_addr},\ \mathrm{pm}'\text{`rw\_addr})(\mathrm{addr}') \\
\{-6\} & \mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}') \\
\{-7\} & \mathrm{pm}'\text{`states}(s_1') \\
\{-8\} & \mathrm{pm}'\text{`states}(s_2') \\
\hline
\{1\} & \mathrm{pdir\_lvl}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}] < 2 \\
\{2\} & \mathrm{Mem?}(\mathrm{xlat\_idx}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}] \\
 & \qquad (\mathrm{pdir\_lvl}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}], \\
 & \qquad\ \mathrm{data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \qquad\qquad (\mathrm{read\_data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \qquad\qquad\quad (\mathrm{pm\_phy},\ \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s_2')\text{`base\_addr}, \\
 & \qquad\ \mathrm{addr}')\text{`type\_of}) \\
 & \quad \wedge \\
 & \quad 0 \le \\
 & \quad\ \mathrm{xlat\_idx}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}] \\
 & \qquad (\mathrm{pdir\_lvl}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}], \\
 & \qquad\ \mathrm{data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \qquad\qquad (\mathrm{read\_data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \qquad\qquad\quad (\mathrm{pm\_phy},\ \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s_2')\text{`base\_addr}, \\
 & \qquad\ \mathrm{addr}')\text{`offset} \\
 & \quad \wedge \\
 & \quad\ \mathrm{xlat\_idx}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}] \\
 & \qquad (\mathrm{pdir\_lvl}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}], \\
 & \qquad\ \mathrm{data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \qquad\qquad (\mathrm{read\_data}[\mathrm{Physical\_memory},\ \mathrm{Pdbr\_type}] \\
 & \qquad\qquad\quad (\mathrm{pm\_phy},\ \mathrm{pdbr\_data\_type})(\mathrm{PDBR})(s_2')\text{`base\_addr}, \\
 & \qquad\ \mathrm{addr}')\text{`offset} \\
 & \quad\ < \mathrm{max\_linear\_offset}
\end{array}
$$

Using lemma xlat_pe_in_pdir_range_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pdir_range_TCC3.1.2`.

`xlat_pe_in_pdir_range_TCC3.2:`

{-1} Mem?(addr′‘type_of)
{-2} $0 \leq$ addr′‘offset
{-3} addr′‘offset $<$ max_linear_offset
{-4} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-5} is_linear_plain_memory?(pm′)
{-6} pm′‘states(s_1')
{-7} pm′‘states(s_2')

{1} OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_2'))
{2} Mem?(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))‘base_addr,
addr′)‘type_of)
∧
$0 \leq$
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))‘base_addr,
addr′)‘offset
∧
xlat_idx[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_2'))‘base_addr,
addr′)‘offset
$<$ max_linear_offset

Using lemma xlat_pe_in_pdir_range_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pdir_range_TCC3.2`.

Q.E.D.

C.117.42 Linear_Memory_Properties.xlat_pe_in_pdir_range

Terse proof for `xlat_pe_in_pdir_range`.

`xlat_pe_in_pdir_range:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, s1, s2: Linear_memory):
       union(pm`ro_addr, pm`rw_addr)(addr) ∧
        is_linear_plain_memory?(pm) ∧ pm`states(s1) ∧ pm`states(s2)
        ⊃
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm`ro_addr ∪ pm`rw_addr)))
                         (xlat_idx(pdir_lvl,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                  (s2))`base_addr,
                                   addr))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pe_in_pdir_range?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`xlat_pe_in_pdir_range.1:`

```
{-1}  Mem?(addr'`type_of)
{-2}  0 ≤ addr'`offset
{-3}  addr'`offset < max_linear_offset
{-4}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states(s1')
{-7}  pm'`states(s2')
--------------------------------------------------------------------------------
{1}  ∃ (lin_a:
            (restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))):
       xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1'))`base_addr,
                lin_a)
        =
        xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2'))`base_addr,
                 addr')
```

Instantiating the top quantifier in 1 with the terms: (addr!1),
we get 2 subgoals:

`xlat_pe_in_pdir_range.1.1:`

```
{-1}  Mem?(addr'`type_of)
{-2}  0 ≤ addr'`offset
{-3}  addr'`offset < max_linear_offset
{-4}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states(s1')
{-7}  pm'`states(s2')
--------------------------------------------------------------------------------
{1}  xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1'))`base_addr, addr')
      =
      xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2'))`base_addr, addr')
```

Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Instantiating the top quantifier in -9 with the terms: (s1!1 s2!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_pe_in_pdir_range.1.1`.
`xlat_pe_in_pdir_range.1.2:`

```
{-1}  Mem?(addr'`type_of)
{-2}  0 ≤ addr'`offset
{-3}  addr'`offset < max_linear_offset
{-4}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states(s'_1)
{-7}  pm'`states(s'_2)
------------------------------------------------------------------------
{1}   restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr))(addr')
```

Expanding the definition of restrict,
which is trivially true.
This completes the proof of `xlat_pe_in_pdir_range.1.2`.
`xlat_pe_in_pdir_range.2:`

```
{-1}  Mem?(addr'`type_of)
{-2}  0 ≤ addr'`offset
{-3}  addr'`offset < max_linear_offset
{-4}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'`states(s'_1)
{-7}  pm'`states(s'_2)
------------------------------------------------------------------------
{1}   OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
```

Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `xlat_pe_in_pdir_range.2`.
Q.E.D.

C.117.43 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC1

Terse proof for `xlat_pe_in_pt_range_TCC1`.
`xlat_pe_in_pt_range_TCC1:`

```
{1}   ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G,
           s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
        pm`states(s_2) ∧
         pm`states(s_1) ∧ is_linear_plain_memory?(pm) ∧ union(pm`ro_addr, pm`rw_addr)(addr)
         ⊃
         OK?[Physical_memory, Pdbr_type]
              (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1))
          ∨
          Exception?[Physical_memory, Pdbr_type]
               (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Instantiating quantified variables,

This completes the proof of `xlat_pe_in_pt_range_TCC1`.
Q.E.D.

C.117.44 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC2

Terse proof for `xlat_pe_in_pt_range_TCC2`.
`xlat_pe_in_pt_range_TCC2`:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G,
s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
pm`states(s_2) ∧
pm`states($s_1$) ∧ is_linear_plain_memory?(pm) ∧ union(pm`ro_addr, pm`rw_addr)(addr)
⊃
OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1))

Repeatedly Skolemizing and flattening,
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
This completes the proof of `xlat_pe_in_pt_range_TCC2`.
Q.E.D.

C.117.45 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC3

Terse proof for `xlat_pe_in_pt_range_TCC3`.
`xlat_pe_in_pt_range_TCC3`:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, access: Memory_access,
priv: Memory_privilege, s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
union(pm`ro_addr, pm`rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧
pm`states(s_1) ∧
pm`states(s_2) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr, addr,
access, priv)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
addr, access, priv)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))))`1
⊃ ptab_lvl[Physical_memory, pm_phy] < max_level

Repeatedly Skolemizing and flattening,
Keeping (2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_pe_in_pt_range_TCC3`.
Q.E.D.

C.117.46 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC4

Terse proof for `xlat_pe_in_pt_range_TCC4`.

xlat_pe_in_pt_range_TCC4:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, access: Memory_access,
priv: Memory_privilege, s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))‘base_addr, addr,
access, priv)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))‘base_addr,
addr, access, priv)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))))‘1
⊃
Mem?(data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))‘base_addr,
addr, access, priv)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))))‘2‘type_of)
∧
0 ≤
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))‘base_addr,
addr, access, priv)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))))‘2‘offset
∧
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))‘base_addr,
addr, access, priv)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))))‘2‘offset
< max_linear_offset

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem max_linear < bits_per_level pe_size max_level

Using lemma xlat_pe_in_pt_range_TCC2,

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Using lemma translate_result_no_leaf,

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC4`.

Q.E.D.

C.117.47 Linear_Memory_Properties.xlat_pe_in_pt_range_TCC5

Terse proof for `xlat_pe_in_pt_range_TCC5`.

xlat_pe_in_pt_range_TCC5:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, access: Memory_access,
priv: Memory_privilege, s_1, s_2: Linear_memory[Physical_memory, pm_phy]):
union(pm`ro_addr, pm`rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧
pm`states(s_1) ∧
pm`states(s_2) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr, addr,
access, priv)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
addr, access, priv)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))))`1
⊃
Mem?(xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
addr, access, priv)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))))`2,
addr)`type_of)
∧
0 ≤
xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
addr, access, priv)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))))`2,
addr)`offset
∧
xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
addr, access, priv)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1))))`2,
addr)`offset
< max_linear_offset

1691

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: Mem max_linear < bits_per_level pe_size max_level

Using lemma xlat_idx_memory_address,

we get 3 subgoals:

xlat_pe_in_pt_range_TCC5.1:

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`2))
⊃
xlat_idx(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`2,
addr′)
< max_linear
{-2} Mem?(addr′`type_of)
{-3} 0 ≤ addr′`offset
{-4} addr′`offset < max_linear_offset
{-5} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′`states(s_1')
{-8} pm′`states(s_2')
{-9} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))

{1} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`1
{2} Mem?(xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`2,
addr′)`type_of)
∧
0 ≤
xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]

Using lemma xlat_idx_memory_address2,

we get 3 subgoals:

xlat_pe_in_pt_range_TCC5.1.1:

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2))
⊃
0 ≤
offset
(xlat_idx(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2,
addr′))
{-2} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2))
⊃
xlat_idx(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2,
addr′)
< max_linear
{-3} Mem?(addr′`type_of) 1695
{-4} 0 ≤ addr′`offset
{-5} addr′`offset < max_linear_offset
{-6} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-7} is_linear_plain_memory?(pm′)
{-8} pm′`states(s'_1)
{-9} pm′`states(s'_2)
{-10} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,

Using lemma translate_result_leaf,

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Using lemma translate_result_no_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.1.1`.

xlat_pe_in_pt_range_TCC5.1.2:

```
{-1}  aligned?(bits_per_level + pe_size)
            (offset
               (data[Physical_memory, [bool, Address]]
                  (translate[Physical_memory, pm_phy]
                     (pdir_lvl[Physical_memory, pm_phy],
                        data[Physical_memory, Pdbr_type]
                           (read_data[Physical_memory, Pdbr_type]
                              (pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                        addr', access', priv')
                     (state[Physical_memory, Pdbr_type]
                        (read_data[Physical_memory, Pdbr_type]
                           (pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2))
      ⊃
      xlat_idx(ptab_lvl[Physical_memory, pm_phy],
               data[Physical_memory, [bool, Address]]
                  (translate[Physical_memory, pm_phy]
                     (pdir_lvl[Physical_memory, pm_phy],
                        data[Physical_memory, Pdbr_type]
                           (read_data[Physical_memory, Pdbr_type]
                              (pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                        addr', access', priv')
                     (state[Physical_memory, Pdbr_type]
                        (read_data[Physical_memory, Pdbr_type]
                           (pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2,
               addr')
        < max_linear
{-2}  Mem?(addr'`type_of)
{-3}  0 ≤ addr'`offset
{-4}  addr'`offset < max_linear_offset
{-5}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-6}  is_linear_plain_memory?(pm')
{-7}  pm'`states(s'_1)
{-8}  pm'`states(s'_2)
{-9}  OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                    addr', access', priv')
                 (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
-------------------------------------------------------------------------
{1}   Mem?(data[Physical_memory, [bool, Address]]
             (translate[Physical_memory, pm_phy]
                (pdir_lvl[Physical_memory, pm_phy],
                   data[Physical_memory, Pdbr_type]
                      (read_data[Physical_memory, Pdbr_type]
                         (pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                   addr', access', priv')
                (state[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2`type_of)
      ∧
      0 ≤
       data[Physical_memory, [bool, Address]]
          (translate[Physical_memory, pm_phy]
             (pdir_lvl[Physical_memory, pm_phy],
                data[Physical_memory, Pdbr_type]
                   (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                addr', access', priv')
             (state[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                   (pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2`offset
       ∧
       data[Physical_memory, [bool, Address]]
          (translate[Physical_memory, pm_phy]
```

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.1.2`.

xlat_pe_in_pt_range_TCC5.1.3:

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`2))
⊃
xlat_idx(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`2,
addr′)
< max_linear
{-2} Mem?(addr′`type_of)
{-3} 0 ≤ addr′`offset
{-4} addr′`offset < max_linear_offset
{-5} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-6} is_linear_plain_memory?(pm′)
{-7} pm′`states(s_1')
{-8} pm′`states(s_2')
{-9} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))

{1} ptab_lvl[Physical_memory, pm_phy] < 2
{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`1
{3} Mem?(xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`2,
addr′)`type_of)
∧
0 ≤
xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.1.3`.

xlat_pe_in_pt_range_TCC5.2:

{-1} Mem?(addr′‘type_of)
{-2} 0 ≤ addr′‘offset
{-3} addr′‘offset < max_linear_offset
{-4} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-5} is_linear_plain_memory?(pm′)
{-6} pm′‘states(s'_1)
{-7} pm′‘states(s'_2)
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))

{1} Mem?(data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘2‘type_of)
∧
0 ≤
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘2‘offset
∧
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘2‘offset
< max_linear_offset
{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
{3} Mem?(xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘2,
addr′)‘type_of)
∧

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.2`.

xlat_pe_in_pt_range_TCC5.3:

{-1} Mem?(addr′`type_of)
{-2} 0 ≤ addr′`offset
{-3} addr′`offset < max_linear_offset
{-4} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-5} is_linear_plain_memory?(pm′)
{-6} pm′`states(s'_1)
{-7} pm′`states(s'_2)
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))

{1} ptab_lvl[Physical_memory, pm_phy] < 2
{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`1
{3} Mem?(xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2,
addr′)`type_of)
∧
0 ≤
xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2,
addr′)`offset
∧
xlat_idx[Physical_memory, pm_phy]
(ptab_lvl[Physical_memory, pm_phy],
data[Physical_memory, [bool, Address]]
(translate[Physical_memory, pm_phy]
(pdir_lvl[Physical_memory, pm_phy],
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
addr′, access′, priv′)
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`2,
addr′)`offset
< max_linear_offset

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range_TCC5.3`.

Q.E.D.

C.117.48 Linear_Memory_Properties.xlat_pe_in_pt_range

Terse proof for `xlat_pe_in_pt_range`.

`xlat_pe_in_pt_range`:

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, access: Mem-
     ory_access,
           priv: Memory_privilege, s1, s2: Linear_memory):
       union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
        is_linear_plain_memory?(pm) ∧
         pm‘states(s1) ∧
          pm‘states(s2) ∧
           OK?(translate(pdir_lvl,
                         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))‘base_addr, addr
                         access, priv)
                        (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))
            ∧
            ¬ data(translate(pdir_lvl,
                             data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))‘base_addr,
                             addr, access, priv)
                            (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1))))‘1
       ⊃
       pe_in_pt_range?(s2,
                       restrict[Address, Memory_Address_4G, boolean]
                           ((pm‘ro_addr ∪ pm‘rw_addr)),
                       ptab_lvl)
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                               data(read_data(pm_phy, pdbr_data_type)
                                                              (PDBR)
                                                              (s1))‘base_addr,
                                               addr, access, priv)
                                              (state(read_data(pm_phy, pdbr_data_type)
                                                              (PDBR)
                                                              (s1))))‘2,
                                addr))
```

Installing automatic rewrites from: (bits_per_level pe_size singleton has_next_state)

Repeatedly Skolemizing and flattening,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`xlat_pe_in_pt_range.1:`

{-1} Mem?(addr′`type_of)
{-2} $0 \leq$ addr′`offset
{-3} addr′`offset $<$ max_linear_offset
{-4} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-5} is_linear_plain_memory?(pm′)
{-6} pm′`states(s_1')
{-7} pm′`states(s_2')
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr, addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
{-9} ptab_lvl = pdir_lvl

{1} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr, addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))`1
{2} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s_1'))`base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s_1'))))`2,
addr′))

Keeping (-9) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.1`.

`xlat_pe_in_pt_range.2:`

{-1} Mem?(addr′‘type_of)
{-2} $0 \leq$ addr′‘offset
{-3} addr′‘offset $<$ max_linear_offset
{-4} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-5} is_linear_plain_memory?(pm′)
{-6} pm′‘states(s'_1)
{-7} pm′‘states(s'_2)
{-8} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))

{1} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
{2} ptab_lvl = pdir_lvl
{3} pe_in_ptab_range?(s'_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))

Copying formula number: -5

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-1 -2 -3 -5 -7 -8),

Case splitting on OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s1!1)), pm!1‘states (state(read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s1!1))),

we get 3 subgoals:

xlat_pe_in_pt_range.2.1:

{-1} $pm'`states$
$(state(read\_data[Physical\_memory, Pdbr\_type](pm\_phy, pdbr\_data\_type)(PDBR)(s_1')))$
{-2} $OK?(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))$
{-3} $\forall (s: (pm'`states)): OK?(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s))$
{-4} $\forall (s_1, s_2: (pm'`states)):$
$pe\_in\_pdir\_range?(s_1,$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$=$
$pe\_in\_pdir\_range?(s_2,$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$\wedge$
$pe\_in\_ptab\_range?(s_1,$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$=$
$pe\_in\_ptab\_range?(s_2,$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
{-5} $plain\_memory?(pm\_phy)$
{-6} $pm'`states = pm\_phy`states$
{-7} $(pm'`other\_actions \subseteq pm\_phy`other\_actions)$
{-8} $(linear\_resolve\_register\_transformers \subseteq pm\_phy`other\_actions)$
{-9} $\forall (a: ((pm'`ro\_addr \cup pm'`rw\_addr))):$
$Mem?(type\_of(a)) \wedge 0 \leq offset(a) \wedge offset(a) < max\_linear\_offset$
{-10} $\forall (s: (pm'`states)):$
$linear\_blessed?(s, restrict[Address, Memory\_Address\_4G, boolean](pm'`ro\_addr),$
$restrict[Address, Memory\_Address\_4G, boolean](pm'`rw\_addr))$
{-11} $Mem?(addr'`type\_of)$
{-12} $0 \leq addr'`offset$
{-13} $addr'`offset < max\_linear\_offset$
{-14} $union(pm'`ro\_addr, pm'`rw\_addr)(addr')$
{-15} $is\_linear\_plain\_memory?(pm')$
{-16} $pm'`states(s_1')$
{-17} $pm'`states(s_2')$
{-18} $OK?(translate(pdir\_lvl, data(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))`base\_addr,$
$addr', access', priv')$
$(state(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))))$

{1} $data(translate(pdir\_lvl, data(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))`base\_addr,$
$addr', access', priv')$
$(state(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))))`1$
{2} $ptab\_lvl = pdir\_lvl$
{3} $pe\_in\_ptab\_range?(s_2',$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$(xlat\_idx(ptab\_lvl,$
$data(translate(pdir\_lvl,$
$data(read\_data(pm\_phy, pdbr\_data\_type)$
$(PDBR)$
$(s_1'))`base\_addr,$
$addr', access', priv')$
$(state(read\_data(pm\_phy, pdbr\_data\_type)$
$(PDBR)$
$(s_1'))))`2,$
$addr'))$

Instantiating the top quantifier in -4 with the terms: (s2!1 state(read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s1!1))),

we get 2 subgoals:

xlat_pe_in_pt_range.2.1.1:

{-1} $pm'`states$
$(state(read\_data[Physical\_memory, Pdbr\_type](pm\_phy, pdbr\_data\_type)(PDBR)(s_1')))$
{-2} $OK?(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))$
{-3} $\forall\ (s\colon (pm'`states)) \colon OK?(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s))$
{-4} $pe\_in\_pdir\_range?(s_2',$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$=$
$pe\_in\_pdir\_range?(state(read\_data[Physical\_memory, Pdbr\_type]$
$(pm\_phy, pdbr\_data\_type)(PDBR)(s_1')),$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$\wedge$
$pe\_in\_ptab\_range?(s_2',$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$=$
$pe\_in\_ptab\_range?(state(read\_data[Physical\_memory, Pdbr\_type]$
$(pm\_phy, pdbr\_data\_type)(PDBR)(s_1')),$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
{-5} $plain\_memory?(pm\_phy)$
{-6} $pm'`states = pm\_phy`states$
{-7} $(pm'`other\_actions \subseteq pm\_phy`other\_actions)$
{-8} $(linear\_resolve\_register\_transformers \subseteq pm\_phy`other\_actions)$
{-9} $\forall\ (a\colon ((pm'`ro\_addr \cup pm'`rw\_addr))\colon$
$Mem?(type\_of(a)) \wedge 0 \leq offset(a) \wedge offset(a) < max\_linear\_offset$
{-10} $\forall\ (s\colon (pm'`states))\colon$
$linear\_blessed?(s, restrict[Address, Memory\_Address\_4G, boolean](pm'`ro\_addr),$
$restrict[Address, Memory\_Address\_4G, boolean](pm'`rw\_addr))$
{-11} $Mem?(addr'`type\_of)$
{-12} $0 \leq addr'`offset$
{-13} $addr'`offset < max\_linear\_offset$
{-14} $union(pm'`ro\_addr, pm'`rw\_addr)(addr')$
{-15} $is\_linear\_plain\_memory?(pm')$
{-16} $pm'`states(s_1')$
{-17} $pm'`states(s_2')$
{-18} $OK?(translate(pdir\_lvl, data(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))`base\_addr,$
$addr', access', priv')$
$(state(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))))$

{1} $data(translate(pdir\_lvl, data(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))`base\_addr,$
$addr', access', priv')$
$(state(read\_data(pm\_phy, pdbr\_data\_type)(PDBR)(s_1'))))`1$
{2} $ptab\_lvl = pdir\_lvl$
{3} $pe\_in\_ptab\_range?(s_2',$
$restrict[Address, Memory\_Address\_4G, boolean]$
$((pm'`ro\_addr \cup pm'`rw\_addr)))$
$(xlat\_idx(ptab\_lvl,$
$data(translate(pdir\_lvl,$
$data(read\_data(pm\_phy, pdbr\_data\_type)$
$(PDBR)$
$(s_1'))`base\_addr,$
$addr', access', priv')$
$(state(read\_data(pm\_phy, pdbr\_data\_type)$
$(PDBR)$
$(s_1'))))`1,$
$addr'))$

Applying disjunctive simplification to flatten sequent,

Replacing using formula -5,

Expanding the definition of pe_in_ptab_range?,

Case splitting on pe_in_pdir_range?[Physical_memory, pm_phy] (state(read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s1!1)), restrict[Address, Memory_Address_4G, boolean] (union[Address](pm!1‘ro_addr, pm!1‘rw_addr))) (xlat_idx[Physical_memory, pm_phy] (pdir_lvl, data[Physical_memory, Pdbr_type] (read_data[Physical_memory, Pdbr_type] (pm_phy, pdbr_data_type)(PDBR)(s1!1))‘base_addr, addr!1)),

we get 3 subgoals:

xlat_pe_in_pt_range.2.1.1.1:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

Instantiating the top quantifier in 3 with the terms: (xlat_idx(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1!1))‘base_addr, addr!1)),

we get 2 subgoals:

xlat_pe_in_pt_range.2.1.1.1.1:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

Instantiating the top quantifier in 3 with the terms: (addr!1),

we get 2 subgoals:

xlat_pe_in_pt_range.2.1.1.1.1.1:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

xlat_pe_in_pt_range.2.1.1.1.1.1.1:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

Using lemma translate_result_as_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.1.1`.

```
xlat_pe_in_pt_range.2.1.1.1.1.1.2:

  {-1}  pe_in_pdir_range?[Physical_memory, pm_phy]
          (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)),
           restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
          (xlat_idx[Physical_memory, pm_phy]
             (pdir_lvl,
              data[Physical_memory, Pdbr_type]
                (read_data[Physical_memory, Pdbr_type]
                   (pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
              addr'))
  {-2}  pm'`states
          (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
  {-3}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
  {-4}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
  {-5}  pe_in_pdir_range?(s'_2,
                          restrict[Address, Memory_Address_4G, boolean]
                            ((pm'`ro_addr ∪ pm'`rw_addr)))
        =
        pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
                                  (pm_phy, pdbr_data_type)(PDBR)(s'_1)),
                          restrict[Address, Memory_Address_4G, boolean]
                            ((pm'`ro_addr ∪ pm'`rw_addr)))
  {-6}  (λ (phy_a: Memory_Address_4G):
           ∃ (pde:
                (pe_in_pdir_range?(s'_2,
                                   restrict[Address, Memory_Address_4G, boolean]
                                     ((pm'`ro_addr ∪ pm'`rw_addr))))):
             ∃ (lin_a:
                  (restrict[Address, Memory_Address_4G, boolean]
                     ((pm'`ro_addr ∪ pm'`rw_addr)))):
               OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s'_2)) ∧
                pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
                                    (s'_2))))
                 ∧
                 xlat_idx(ptab_lvl,
                          base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
                                      (s'_2))),
                          lin_a)
                  = phy_a)
        =
        (λ (phy_a: Memory_Address_4G):
           ∃ (pde:
                (pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
                                           (pm_phy, pdbr_data_type)(PDBR)(s'_1)),
                                   restrict[Address, Memory_Address_4G, boolean]
                                     ((pm'`ro_addr ∪ pm'`rw_addr))))):
             ∃ (lin_a:
                  (restrict[Address, Memory_Address_4G, boolean]
                     ((pm'`ro_addr ∪ pm'`rw_addr)))):
               OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
                     (state(read_data[Physical_memory, Pdbr_type]
                              (pm_phy, pdbr_data_type)(PDBR)(s'_1)))) ∧
                pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
                                    (state(read_data[Physical_memory, Pdbr_type]
                                             (pm_phy, pdbr_data_type)(PDBR)
                                             (s'_1)))))) ∧
                 xlat_idx(ptab_lvl,
                          base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
                                      (state(read_data[Physical_memory, Pdbr_type]
                                               (pm_phy, pdbr_data_type)(PDBR)
                                               (s'_1))))),
                          lin_a)
```

Expanding the definition of translate,

Installing automatic rewrites from: (##! fatal_result! raise_fault! exception_result! ok_result!)

Expanding the definition of paging_type?,

Expanding the definition of pdir_entry?,

Expanding the definition of present?,

Expanding the definition of is_leaf?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.1.2`.

xlat_pe_in_pt_range.2.1.1.1.1.1.3:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

Keeping (-20 3) and hiding *,

Expanding the definition of translate,

Installing automatic rewrites from: (##! fatal_result! raise_fault! exception_result! ok_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.1.3`.

xlat_pe_in_pt_range.2.1.1.1.1.2:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.1.2`.

xlat_pe_in_pt_range.2.1.1.1.2:

{-1} pe_in_pdir_range?[Physical_memory, pm_phy]
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
(xlat_idx[Physical_memory, pm_phy]
(pdir_lvl,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))`base_addr,
addr'))

{-2} pm'`states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))

{-3} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{-4} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-5} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))

{-6} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm'`ro_addr ∪ pm'`rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)

which is trivially true.

This completes the proof of `xlat_pe_in_pt_range.2.1.1.1.2`.

xlat_pe_in_pt_range.2.1.1.2:

{-1} pm′‘states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))
{-2} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))
{-3} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-4} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-5} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)
= phy_a)
{-6} plain_memory?(pm_phy)
{-7} pm′‘states = pm_phy‘states
{-8} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-9} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-10} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-11} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),

Hiding formulas: 4,

Using lemma xlat_pe_in_pdir_range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.1.1.2`.

xlat_pe_in_pt_range.2.1.1.3:

{-1} pm′‘states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s_1')))
{-2} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))
{-3} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-4} pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-5} (λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(s_2',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)(s_2')) ∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(s_2'))),
lin_a)
= phy_a)
=
(λ (phy_a: Memory_Address_4G):
∃ (pde:
(pe_in_pdir_range?(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1')),
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr))))):
∃ (lin_a:
(restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))):
OK?(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
pde_pt?(pdir(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))))
∧
xlat_idx(ptab_lvl,
base(data(read_data(pm_phy, paging_data_type(pdir_lvl))(pde)
(state(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(s_1'))))),
lin_a)
= phy_a)
{-6} plain_memory?(pm_phy)
{-7} pm′‘states = pm_phy‘states
{-8} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-9} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-10} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-11} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),

Using lemma xlat_idx_memory_address,
Using lemma xlat_idx_memory_address2,
Installing automatic rewrites from: max_linear < Mem <= min_linear
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_pe_in_pt_range.2.1.1.3`.
`xlat_pe_in_pt_range.2.1.2`:

```
{-1}  pm'`states
        (state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-3}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-4}  plain_memory?(pm_phy)
{-5}  pm'`states = pm_phy`states
{-6}  (pm'`other_actions ⊆ pm_phy`other_actions)
{-7}  (linear_resolve_register_transformers ⊆ pm_phy`other_actions)
{-8}  ∀ (a: ((pm'`ro_addr ∪ pm'`rw_addr))):
        Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-9}  ∀ (s: (pm'`states)):
        linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
                           restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
{-10} Mem?(addr'`type_of)
{-11} 0 ≤ addr'`offset
{-12} addr'`offset < max_linear_offset
{-13} union(pm'`ro_addr, pm'`rw_addr)(addr')
{-14} is_linear_plain_memory?(pm')
{-15} pm'`states(s'_1)
{-16} pm'`states(s'_2)
{-17} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                    addr', access', priv')
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
-------------------------------------------------------------------------------
{1}   pm'`states
        (state[Physical_memory, Pdbr_type]
          (read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{2}   data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))`base_addr,
                     addr', access', priv')
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))`1
{3}   ptab_lvl = pdir_lvl
{4}   pe_in_ptab_range?(s'_2,
                        restrict[Address, Memory_Address_4G, boolean]
                          ((pm'`ro_addr ∪ pm'`rw_addr)))
                      (xlat_idx(ptab_lvl,
                                data(translate(pdir_lvl,
                                               data(read_data(pm_phy, pdbr_data_type)
                                                              (PDBR)
                                                              (s'_1))`base_addr,
                                               addr', access', priv')
                                             (state(read_data(pm_phy, pdbr_data_type)
                                                             (PDBR)
                                                             (s'_1))))`2,
                                addr'))
```

which is trivially true.
This completes the proof of `xlat_pe_in_pt_range.2.1.2`.

xlat_pe_in_pt_range.2.2:

{-1} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))
{-2} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-3} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-4} plain_memory?(pm_phy)
{-5} pm′‘states = pm_phy‘states
{-6} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-7} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-8} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-9} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-10} Mem?(addr′‘type_of)
{-11} 0 ≤ addr′‘offset
{-12} addr′‘offset < max_linear_offset
{-13} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-14} is_linear_plain_memory?(pm′)
{-15} pm′‘states(s'_1)
{-16} pm′‘states(s'_2)
{-17} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))

{1} pm′‘states
(state(read_data[Physical_memory, Pdbr_type](pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
{3} ptab_lvl = pdir_lvl
{4} pe_in_ptab_range?(s'_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, priv′)
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_pe_in_pt_range.2.2`.

xlat_pe_in_pt_range.2.3:

{-1} $\forall$ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-2} $\forall$ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1, restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2, restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1, restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2, restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))

{-3} plain_memory?(pm_phy)

{-4} pm′‘states = pm_phy‘states

{-5} (pm′‘other_actions ⊆ pm_phy‘other_actions)

{-6} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)

{-7} $\forall$ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset

{-8} $\forall$ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))

{-9} Mem?(addr′‘type_of)

{-10} 0 ≤ addr′‘offset

{-11} addr′‘offset < max_linear_offset

{-12} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)

{-13} is_linear_plain_memory?(pm′)

{-14} pm′‘states(s_1')

{-15} pm′‘states(s_2')

{-16} OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))‘base_addr, addr′, access′, priv′)(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))

{1} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))

{2} data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))‘base_addr, addr′, access′, priv′)(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))‘1

{3} ptab_lvl = pdir_lvl

{4} pe_in_ptab_range?(s_2', restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))(xlat_idx(ptab_lvl, data(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))‘base_addr, addr′, access′, priv′)(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))‘2, addr′))

Instantiating quantified variables,
This completes the proof of `xlat_pe_in_pt_range.2.3`.
Q.E.D.

C.117.49 Linear_Memory_Properties.linear_resolve_unchanged_pm_phy

Terse proof for `linear_resolve_unchanged_pm_phy`.
`linear_resolve_unchanged_pm_phy:`

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, access: Memory_access):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧ is_linear_plain_memory?(pm) ∧ pm‘states(s) ⊃
unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm‘states,
singleton(expr_2_super[Physical_memory, Address](linear_resolve(addr, access))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, F

Installing automatic rewrites from: plain_memory_unchanged_memory_ok_result
Repeatedly Skolemizing and flattening,
Letting addresses name difference(union(pm_phy‘ro_addr, pm_phy‘rw_addr), extend[Address, Memory_Address_4G, bool, FALSE] (address_in_pt_range?(s!1, restrict [Address, Memory_Address_4G, boolean] (union (pm!1‘ro_addr, pm!1‘rw_addr))))),
Replacing using formula -1,
Case splitting on subset?(addresses, union(pm_phy‘ro_addr, pm_phy‘rw_addr)),
we get 2 subgoals:
`linear_resolve_unchanged_pm_phy.1:`

{-1} (addresses ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](add
= addresses
{-3} Mem?(addr′‘type_of)
{-4} 0 ≤ addr′‘offset
{-5} addr′‘offset < max_linear_offset
{-6} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s′)

{1} unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address](linear_resolve(addr′, access′))),
addresses)

Expanding the definition of linear_resolve,
Using lemma pm_plain_phy,
Using lemma pm_states,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Installing automatic rewrites from: (has_next_state! max_level! bits_per_level! bus_width! pe_size! singleton! <! Mem! max_linear!)
Copying formula number: -9
Expanding the definition of is_linear_plain_memory?,
Applying disjunctive simplification to flatten sequent,

Installing automatic rewrites from: pdir_lvl!! ptab_lvl!!

Case splitting on FORALL (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool, base: {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl[Physical_memory, pm_phy] * bits_per_level [Physical_memory, pm_phy]) - bits_per_level [Physical_memory, pm_phy] + bus_width) (offset(b)) ELSE aligned?(bits_per_level [Physical_memory, pm_phy] + pe_size) (offset(b)) ENDIF}): NOT leaf IMPLIES (FORALL (x1: Physical_memory): every[Physical_memory, [bool, Address]] (LAMBDA (x: Physical_memory): TRUE, LAMBDA (t: [bool, Address]): Mem?(t`2`type_of) AND 0 <= t`2`offset AND t`2`offset < max_linear_offset) (translate[Physical_memory, pm_phy] (1, base, addr!1, access!1, segment_to_priv(cs))(x1))), FORALL (cs: Segment_Reg_type, pdbr: Pdbr_type): FORALL (x1: Physical_memory): every[Physical_memory, [bool, Address]] (LAMBDA (x: Physical_memory): TRUE, LAMBDA (t: [bool, Address]): Mem?(t`2`type_of) AND (0 <= t`2`offset AND t`2`offset < max_linear_offset) AND IF t`1 THEN aligned?(22)(offset(t`2)) ELSE aligned?(12)(offset(t`2)) ENDIF) (translate[Physical_memory, pm_phy] (0, pdbr`base_addr, addr!1, access!1, segment_to_priv(cs)) (x1)),

we get 5 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1:

{-1}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x₁: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x₁))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x₁: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr′, access′, segment_to_priv(cs))(x₁)))
{-3}  pm′‘mem = linear_pm
{-4}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-7}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₁)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₂))
{-8}  ∀ (s₁, s₂: (pm′‘states)):
        pe_in_pdir_range?(s₁,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s₂,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s₁,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s₂,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-9}  ∀ (s: (pm′‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                              lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10} ∀ (s₁, s₂: (pm′‘states), lvl: Level,
```

Rewriting using plain_memory_unchanged_composition [Physical_memory, Segment_Reg_type, Address], matching in * where P gets LAMBDA (t: Segment_Reg_type): FORALL (s: (pm!1‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) AND t = data(read_data(pm_phy, segment_reg_data_type)(CS)(s)),

we get 3 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.1:

{-1}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x₁: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x₁))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x₁: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr′, access′, segment_to_priv(cs))(x₁)))
{-3}  pm′‘mem = linear_pm
{-4}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-7}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₁)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₂))
{-8}  ∀ (s₁, s₂: (pm′‘states)):
        pe_in_pdir_range?(s₁,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s₂,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s₁,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s₂,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-9}  ∀ (s: (pm′‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                              lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10} ∀ (s₁, s₂: (pm′‘states), lvl: Level,
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Keeping (-7 -8 1) and hiding *,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.1`.

```
linear_resolve_unchanged_pm_phy.1.1.2:

{-1}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x₁: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x₁))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                             (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x₁: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr′, access′, segment_to_priv(cs))(x₁)))
{-3}  pm′‘mem = linear_pm
{-4}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-7}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₁)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₂))
{-8}  ∀ (s₁, s₂: (pm′‘states)):
        pe_in_pdir_range?(s₁,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s₂,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s₁,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s₂,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-9}  ∀ (s: (pm′‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                              lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10} ∀ (s₁, s₂: (pm′‘states), lvl: Level,
```

Hiding formulas: 2,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_mono,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.2`.

`linear_resolve_unchanged_pm_phy.1.1.3:`

```
{-1}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x₁: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x₁))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                             (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x₁: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr′, access′, segment_to_priv(cs))(x₁)))
{-3}  pm′‘mem = linear_pm
{-4}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s₁)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s₂))
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-7}  ∀ (s₁, s₂: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₁)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s₂))
{-8}  ∀ (s₁, s₂: (pm′‘states)):
        pe_in_pdir_range?(s₁,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s₂,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s₁,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s₂,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-9}  ∀ (s: (pm′‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                              lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-10} ∀ (s₁, s₂: (pm′‘states), lvl: Level,
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_unchanged_composition[Physical_memory, Pdbr_type, Address], matching in * where P gets LAMBDA (base: Pdbr_type): FORALL (s: (pm_phy‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) AND data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = base,

we get 5 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.1:

{-1}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                     IF t‘1
                       THEN aligned?(22)(offset(t‘2))
                     ELSE aligned?(12)(offset(t‘2))
                     ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
             every[Physical_memory, [bool, Address]]
                  (λ (x: Physical_memory): TRUE,
                     λ (t: [bool, Address]):
                       Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                  (translate[Physical_memory, pm_phy]
                            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'‘mem = linear_pm
{-5}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-8}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-9}  ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
1744      pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-10} ∀ (s: (pm'‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'‘ro_addr ∪ pm'‘rw_addr)),
```

Keeping (-7 -8 1) and hiding *,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.2:

{-1}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                     IF t‘1
                       THEN aligned?(22)(offset(t‘2))
                     ELSE aligned?(12)(offset(t‘2))
                     ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                             (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
             every[Physical_memory, [bool, Address]]
                  (λ (x: Physical_memory): TRUE,
                     λ (t: [bool, Address]):
                       Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                  (translate[Physical_memory, pm_phy]
                            (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'‘mem = linear_pm
{-5}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-8}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-9}  ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
1746      pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-10} ∀ (s: (pm'‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'‘ro_addr ∪ pm'‘rw_addr)),
```

Hiding formulas: 2,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Using lemma unchanged_memory_invariant_mono,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.2`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3:

{-1}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                     IF t‘1
                       THEN aligned?(22)(offset(t‘2))
                     ELSE aligned?(12)(offset(t‘2))
                     ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                             (offset(b))
                        ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-4}  pm'‘mem = linear_pm
{-5}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-7}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-8}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-9}  ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
1748      pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-10} ∀ (s: (pm'‘states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'‘ro_addr ∪ pm'‘rw_addr)),
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_unchanged_composition [Physical_memory, [leaf: bool, {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl * bits_per_level) - bits_per_level + bus_width)(offset(b)) ELSE aligned?(bits_per_level + pe_size)(offset(b)) ENDIF}], Address], matching in * where P gets LAMBDA (d: [leaf: bool, {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl * bits_per_level) - bits_per_level + bus_width)(offset(b)) ELSE aligned?(bits_per_level + pe_size)(offset(b)) ENDIF}]): FORALL (s: (pm_phy`states)): OK?(translate(0, d!2`base_addr, addr!1, access!1, segment_to_priv(d!1))(s)) AND d = data(translate(0, d!2`base_addr, addr!1, access!1, segment_to_priv(d!1))(s)),

we get 4 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.1:

{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                             (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm'‘mem = linear_pm
{-6}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-11} ∀ (s: (pm'‘states), lvl: Level,
           a:
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Case splitting on pe_in_pt_range?(s!2, restrict[Address, Memory_Address_4G, boolean] (union(pm!1`ro_addr, pm!1`rw_addr)), 0) (xlat_idx(0, d!2`base_addr, addr!1)),

we get 3 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.1.1:

{-1}  pe_in_pt_range?(s'',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'`ro_addr ∪ pm'`rw_addr)),
                      0)
                     (xlat_idx(0, d''`base_addr, addr'))
{-2}  pm_phy`states(s''')
{-3}  pm_phy`states(s'')
{-4}  OK?(translate(0, d''`base_addr, addr', access', segment_to_priv(d'))(s''))
{-5}  ∀ (s: (pm_phy`states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-6}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t`2`type_of) ∧
                     (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                      IF t`1
                        THEN aligned?(22)(offset(t`2))
                      ELSE aligned?(12)(offset(t`2))
                      ENDIF)
                (translate[Physical_memory, pm_phy]
                          (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x1))
{-8}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                      IF leaf
                        THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                            (offset(b))
                      ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                   (offset(b))
                      ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-9}  pm'`mem = linear_pm
{-10} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-12} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-13} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-14} ∀ (s1, s2: (pm'`states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                            ((pm'`ro_addr ∪ pm'`rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                             ((pm'`ro_addr ∪ pm'`rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
```

Using lemma translate_same_result,

Using lemma translate_result_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.1.2:

{-1}  pm_phy‘states(s‴)
{-2}  pm_phy‘states(s″)
{-3}  OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s″))
{-4}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-5}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-7}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                          ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-8}  pm′‘mem = linear_pm
{-9}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-13} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
```

Hiding formulas: 2,

Expanding the definition of pe_in_pt_range?,

Expanding the definition of pe_in_pdir_range?,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in 1 with the terms: addr′,

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.1:

{-1}  0 = 0
{-2}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s''))
{-3}  pm_phy`states(s''')
{-4}  pm_phy`states(s'')
{-5}  OK?(translate(0, d''`base_addr, addr', access', segment_to_priv(d'))(s''))
{-6}  data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'')) = d''
{-7}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x_1: Physical_memory):
          every[Physical_memory, [bool, Address]]
              (λ (x: Physical_memory): TRUE,
                λ (t: [bool, Address]):
                  Mem?(t`2`type_of) ∧
                   (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                    IF t`1
                      THEN aligned?(22)(offset(t`2))
                    ELSE aligned?(12)(offset(t`2))
                    ENDIF)
              (translate[Physical_memory, pm_phy]
                   (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x_1))
{-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                   IF leaf
                     THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                    (offset(b))
                   ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                 (offset(b))
                   ENDIF}):
        ¬ leaf ⊃
         (∀ (x_1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
                (translate[Physical_memory, pm_phy]
                     (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-10} pm'`mem = linear_pm
{-11} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s_1, s_2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-13} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s_1, s_2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-15} ∀ (s_1, s_2: (pm'`states)):
        ((λ (phy_a: Memory_Address_4G):
            OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) ∧
             (∃ (lin_a:
                   (restrict[Address, Memory_Address_4G, boolean]
                        ((pm'`ro_addr ∪ pm'`rw_addr)))):
                xlat_idx(0, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
                         lin_a)
                 = phy_a))
          =
          (λ (phy_a: Memory_Address_4G):
             OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2)) ∧
              (∃ (lin_a:
                    (restrict[Address, Memory_Address_4G, boolean]
                         ((pm'`ro_addr ∪ pm'`rw_addr)))):
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.2:

```
{-1}  0 = 0
{-2}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s''))
{-3}  pm_phy`states(s''')
{-4}  pm_phy`states(s'')
{-5}  OK?(translate(0, d''`base_addr, addr', access', segment_to_priv(d'))(s''))
{-6}  data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'')) = d''
{-7}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x_1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t`2`type_of) ∧
                     (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                      IF t`1
                        THEN aligned?(22)(offset(t`2))
                      ELSE aligned?(12)(offset(t`2))
                      ENDIF)
               (translate[Physical_memory, pm_phy]
                         (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x_1))
{-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                          ENDIF}):
         ¬ leaf ⊃
          (∀ (x_1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr', access', segment_to_priv(cs))(x_1)))
{-10} pm'`mem = linear_pm
{-11} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s_1, s_2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-13} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s_1, s_2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-15} ∀ (s_1, s_2: (pm'`states)):
        ((λ (phy_a: Memory_Address_4G):
              OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) ∧
               (∃ (lin_a:
                        (restrict[Address, Memory_Address_4G, boolean]
                                 ((pm'`ro_addr ∪ pm'`rw_addr)))):
                  xlat_idx(0, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1))`base_addr,
                           lin_a)
                   = phy_a))
          =
          (λ (phy_a: Memory_Address_4G):
               OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2)) ∧
                (∃ (lin_a:
                         (restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'`ro_addr ∪ pm'`rw_addr)))):
```

Keeping (-29 1) and hiding *,

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.2.2`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.1.3:

{-1}  pm_phy‘states(s‴)
{-2}  pm_phy‘states(s″)
{-3}  OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s″))
{-4}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-5}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-7}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                          ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-8}  pm′‘mem = linear_pm
{-9}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-13} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
```

Hiding formulas: 2,

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.1.3`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.2:`

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Hiding formulas: 2,

Using lemma translate_unchanged_pm_phy_except_pe,

Simplifying, rewriting, and recording with decision procedures,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 5 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.1:

{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(0,
                                                                   d″‘base_addr,
                                                                   addr′,
                                                                   access′,
                                                                   seg-
      ment_to_priv(d′)))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(0,
                                                                   d″‘base_addr,
                                                                   addr′,
                                                                   access′,
                                                                   seg-
      ment_to_priv(d′)))),
                                  addresses)
{-3}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-4}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
               (translate[Physical_memory, pm_phy]
                         (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                   IF leaf
                     THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                         (offset(b))
                   ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                 (offset(b))
                   ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-7}  pm′‘mem = linear_pm
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
```

Case splitting on Transformer_Super_Embedding_Expr [Physical_memory, [bool, Address_Helpers.Address]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (0, d!2`base_addr, addr!1, access!1, Segment_Types.segment_to_priv(d!1))) = Transformer_Super_Embedding_Expr [Physical_memory, [leaf: bool, {b: Memory_Address_4G | IF leaf THEN aligned?(-1 * (pdir_lvl[Physical_memory, pm_phy] * bits_per_level [Physical_memory, pm_phy]) - bits_per_level[Physical_memory, pm_phy] + bus_width) (offset(b)) ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size) (offset(b)) ENDIF}]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (0, d!2`base_addr, addr!1, access!1, Segment_Types.segment_to_priv(d!1))),

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.1:

{-1}  Transformer_Super_Embedding_Expr[Physical_memory, [bool, Ad-
      dress_Helpers.Address]].expr_2_super
          (Linear_Memory[Physical_memory, pm_phy].translate
              (0, d''`base_addr, addr', access', Segment_Types.segment_to_priv(d')))
       =
      Transformer_Super_Embedding_Expr
          [Physical_memory,
            [leaf: bool,
              {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_le
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                              (offset(b))
                          ENDIF}]].expr_2_super
          (Linear_Memory[Physical_memory, pm_phy].translate
              (0, d''`base_addr, addr', access', Segment_Types.segment_to_priv(d')))
{-2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(translate(0,
                                                                   d''`base_addr,
                                                                   addr',
                                                                   access',
                                                                   seg-
      ment_to_priv(d')))),
                                  ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(
{-3}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(translate(0,
                                                                   d''`base_addr,
                                                                   addr',
                                                                   access',
                                                                   seg-
      ment_to_priv(d')))),
                                  addresses)
{-4}  ∀ (s: (pm_phy`states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-5}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x_1: Physical_memory):
          every[Physical_memory, [bool, Address]]
              (λ (x: Physical_memory): TRUE,
                λ (t: [bool, Address]):
                  Mem?(t`2`type_of) ∧
                   (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                    IF t`1
                      THEN aligned?(22)(offset(t`2))
                    ELSE aligned?(12)(offset(t`2))
                    ENDIF)
              (translate[Physical_memory, pm_phy]
                  (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x_1))
{-7}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
          base:
            {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                            (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x_1: Physical_memory):
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.2:

{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(0,
                                                                   d″‘base_addr,
                                                                   addr′,
                                                                   access′,
                                                                   seg-
      ment_to_priv(d′)))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(0,
                                                                   d″‘base_addr,
                                                                   addr′,
                                                                   access′,
                                                                   seg-
      ment_to_priv(d′)))),
                                  addresses)
{-3}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-4}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
               (translate[Physical_memory, pm_phy]
                         (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                      IF leaf
                        THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                            (offset(b))
                      ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                    (offset(b))
                      ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-7}  pm′‘mem = linear_pm
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
```

Keeping (1) and hiding *,

Applying extensionality,

Expanding the definition(s) of (expr_2_super expr_2_super_res has_next_state),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.2:

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
              (λ (x: Physical_memory): TRUE,
                 λ (t: [bool, Address]):
                   Mem?(t‘2‘type_of) ∧
                    (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                     IF t‘1
                       THEN aligned?(22)(offset(t‘2))
                     ELSE aligned?(12)(offset(t‘2))
                     ENDIF)
             (translate[Physical_memory, pm_phy]
                    (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                    IF leaf
                      THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                          (offset(b))
                    ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                          (offset(b))
                    ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                (λ (x: Physical_memory): TRUE,
                   λ (t: [bool, Address]):
                     Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                (translate[Physical_memory, pm_phy]
                      (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Hiding formulas: (1 3),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -14 with the terms: (s1!1 s2!1 0 xlat_idx(0, d!2`base_addr, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.1:

{-1} pm′‘states(s'_1)
{-2} pm′‘states(s'_2)
{-3} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-4} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-6} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-7} pm′‘mem = linear_pm
{-8} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-10} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-12} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-13} ∀ (

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.1`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.2:`

{-1} pm′‘states(s'_1)
{-2} pm′‘states(s'_2)
{-3} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-4} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-6} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-7} pm′‘mem = linear_pm
{-8} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-10} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-12} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-13} ∀ (

Adding type constraints for d!2‘base_addr,

Instantiating the top quantifier in -7 with the terms: (s1!1),

Using lemma xlat_pe_in_pdir_range,

Expanding the definition of pe_in_pt_range?,

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.3:

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                         IF leaf
                           THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                         ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                         ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Hiding formulas: (1 3),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -12 with the terms: (s!2 0 xlat_idx(0, d!2`base_addr, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.1:

```
{-1}  pm'‘states(s'')
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
                (translate[Physical_memory, pm_phy]
                          (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                          ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6}  pm'‘mem = linear_pm
{-7}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                    ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-12} OK?(read_data(pm_phy, paging_data_type(0))(xlat_idx(0, d'''‘base_addr, addr'))(s''))
```

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.1`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.2:`

```
{-1}  pm'‘states(s'')
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
                (translate[Physical_memory, pm_phy]
                          (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                          ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6}  pm'‘mem = linear_pm
{-7}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                    ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-12} ∀ (s1, s2: (pm'‘states), lvl: Level,
```

Adding type constraints for d!2‘base_addr,

Instantiating quantified variables,

Using lemma xlat_pe_in_pdir_range,

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.3.2`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.2.4:`

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Using lemma subset_ptab_address_rw_addr,

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1:

{-1}  (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, bool](pm'`ro_addr) ∪ restrict
       ∧
       (address_block(xlat_idx(0, d''`base_addr, addr'), expt(2, pe_size)) ⊆ extend [Address, Memor
       ⊃
       (address_block(xlat_idx(0, d''`base_addr, addr'), expt(2, pe_size)) ⊆ pm_phy`rw_addr)
{-2}  ∀ (s: (pm_phy`states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t`2`type_of) ∧
                     (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                      IF t`1
                        THEN aligned?(22)(offset(t`2))
                      ELSE aligned?(12)(offset(t`2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                        (offset(b))
                          ENDIF}):
         ¬ leaf ⊃
          (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6}  pm'`mem = linear_pm
{-7}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm'`states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'`ro_addr ∪ pm'`rw_addr)))
         =
1784     pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'`ro_addr ∪ pm'`rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'`ro_addr ∪ pm'`rw_addr)))
          =
```

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: (2 3 4),

Applying propositional simplification,

we get 2 subgoals:

`linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1:`

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                    ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Using lemma pm_address_in_pt_range_in_rw_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)),

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.1:

{-1}  restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr)) =
        (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_A
{-2}  is_linear_plain_memory?(pm')
{-3}  (address_in_pt_range?(s', restrict [Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'
{-4}  ∀ (s: (pm_phy'states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-5}  ∀ (s: (pm''states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t'2'type_of) ∧
                     (0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset) ∧
                      IF t'1
                        THEN aligned?(22)(offset(t'2))
                      ELSE aligned?(12)(offset(t'2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr'base_addr, addr', access', segment_to_priv(cs))(x1))
{-7}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                          IF leaf
                            THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                                  (offset(b))
                          ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                          (offset(b))
                          ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t'2'type_of) ∧ 0 ≤ t'2'offset ∧ t'2'offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-8}  pm''mem = linear_pm
{-9}  ∀ (s: (pm''states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (s1, s2: (pm''states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-11} ∀ (s: (pm''states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s1, s2: (pm''states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-13} ∀ (s1, s2: (pm''states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'ro_addr ∪ pm'rw_addr)))
         =
         pe_in_pdir_range?(s2,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'ro_addr ∪ pm'rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'ro_addr ∪ pm'rw_addr)))
          =
          pe_in_ptab_range?(s2,
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.2:

{-1}  is_linear_plain_memory?(pm′)
{-2}  (address_in_pt_range?(s′, restrict [Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘
{-3}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-4}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-7}  pm′‘mem = linear_pm
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.1.2`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2:

{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                    ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.1:

{-1}  pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, bool]
                        ((pm'`ro_addr ∪ pm'`rw_addr)),
                      0)
                     (xlat_idx(0, d''`base_addr, addr'))
      ⊃
      (address_block(xlat_idx(0, d''`base_addr, addr'), expt(2, pe_size)) ⊆ extend [Address, Memor
{-2}  ∀ (s: (pm_phy`states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-3}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t`2`type_of) ∧
                     (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                      IF t`1
                        THEN aligned?(22)(offset(t`2))
                      ELSE aligned?(12)(offset(t`2))
                      ENDIF)
               (translate[Physical_memory, pm_phy]
                         (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                         IF leaf
                           THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                               (offset(b))
                         ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                       (offset(b))
                         ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
                             (1, base, addr', access', segment_to_priv(cs))(x1)))
{-6}  pm'`mem = linear_pm
{-7}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm'`states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                            ((pm'`ro_addr ∪ pm'`rw_addr)))
1794
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                             ((pm'`ro_addr ∪ pm'`rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
```

Using lemma xlat_pe_in_pdir_range,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.1`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.2:`

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                    ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.2:

```
{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-2}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-5}  pm′‘mem = linear_pm
{-6}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm′‘ro_addr ∪ pm′‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-11} ∀ (s: (pm′‘states), lvl: Level,
           a:
```

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.4.2`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.5:

{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(0,
                                                                   d″‘base_addr,
                                                                   addr′,
                                                                   access′,
                                                                   segment_to_priv(d′)))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-3}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                              (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-6}  pm′‘mem = linear_pm
{-7}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
```

Replacing using formula -21,

Rewriting using subset_of_differences, matching in *,

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.1:

{-1}  pe_in_pt_range?(s',
                      restrict[Address, Memory_Address_4G, boolean]
                        ((pm'`ro_addr ∪ pm'`rw_addr)),
                      0)
                     (xlat_idx(0, d''`base_addr, addr'))
       ⊃
       (address_block(xlat_idx(0, d''`base_addr, addr'), expt(2, pe_size)) ⊆ extend [Address, Memor
{-2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  singleton(expr_2_super(translate(0,
                                                                   d''`base_addr,
                                                                   addr',
                                                                   access',
                                                                   seg-
      ment_to_priv(d'))))),
                                  ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(
{-3}  ∀ (s: (pm_phy`states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-4}  ∀ (s: (pm'`states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t`2`type_of) ∧
                     (0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
                      IF t`1
                        THEN aligned?(22)(offset(t`2))
                      ELSE aligned?(12)(offset(t`2))
                      ENDIF)
               (translate[Physical_memory, pm_phy]
                         (0, pdbr`base_addr, addr', access', segment_to_priv(cs))(x1))
{-6}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr', access', segment_to_priv(cs))(x1)))
{-7}  pm'`mem = linear_pm
{-8}  ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-10} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-11} ∀ (s1, s2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-12} ∀ (s1, s2: (pm'`states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
```

Using lemma xlat_pe_in_pdir_range,

Expanding the definition of pe_in_pt_range?,

Instantiating the top quantifier in -4 with the terms: (s!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.2:

{-1}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(0,
                                                                   d″‘base_addr,
                                                                   addr′,
                                                                   access′,
                                                                   segment_to_priv(d′)))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(
{-2}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-3}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-5}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                              (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-6}  pm′‘mem = linear_pm
{-7}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-9}  ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-10} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-11} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
```

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.2.5.2`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.3:

{-1}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-2}  ∀ (s: (pm'‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
              (translate[Physical_memory, pm_phy]
                        (0, pdbr‘base_addr, addr', access', segment_to_priv(cs))(x1))
{-4}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                      (offset(b))
                        ENDIF}):
         ¬ leaf ⊃
          (∀ (x1: Physical_memory):
               every[Physical_memory, [bool, Address]]
                    (λ (x: Physical_memory): TRUE,
                       λ (t: [bool, Address]):
                         Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                    (translate[Physical_memory, pm_phy]
                              (1, base, addr', access', segment_to_priv(cs))(x1)))
{-5}  pm'‘mem = linear_pm
{-6}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-8}  ∀ (s: (pm'‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-9}  ∀ (s1, s2: (pm'‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-10} ∀ (s1, s2: (pm'‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'‘ro_addr ∪ pm'‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm'‘ro_addr ∪ pm'‘rw_addr)))
          =
          pe_in_ptab_range?(s2,
                            restrict[Address, Memory_Address_4G, boolean]
                                    ((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-11} ∀ (s: (pm'‘states), lvl: Level,
           a:
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using plain_memory_unchanged_composition [Physical_memory, [bool, Memory_Address_4G], Address], matching in * where P gets LAMBDA (d: [bool, Memory_Address_4G]): TRUE,

Hiding formulas: 3,

Case splitting on aligned?(12)(offset(d!3‘2)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1:

{-1} aligned?(12)(offset(d'''`2))

{-2} ∀ (s: (pm_phy`states)):
OK?(translate(0, d'''`base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d'''`base_addr, addr′, access′, segment_to_priv(d'))(s))

{-3} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''

{-4} ∀ (s: (pm′`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-5} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr′, access′, segment_to_priv(cs))(x_1))

{-6} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))

{-7} pm′`mem = linear_pm

{-8} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-9} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))

{-10} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-11} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))

{-12} ∀ (s_1, s_2: (pm′`states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_ptab_range?(s_2,

Using lemma unchanged_memory_invariant_mono,

Using lemma xlat_idx_memory_address,

Using lemma translate_unchanged_pm_phy_except_pe,

Simplifying, rewriting, and recording with decision procedures,

Case splitting on subset?(addresses, difference(union(pm_phy‘ro_addr, pm_phy‘rw_addr), address_block(xlat_idx(1, d!3‘2, addr!1), expt(2, 2)))),

we get 2 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1:

{-1}  (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), exp
{-2}  ((address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2)) ⊆ pm_phy‘rw_addr) ∧
        (∀ (s: (pm′‘states)):
           OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d‴‘2, addr′))(s)))
         ∧
         (∀ (s1, s2: (pm′‘states)):
            set_reference(data(read_data(pm_phy, paging_data_type(1))
                                        (xlat_idx(1, d‴‘2, addr′))(s1)),
                          Write)
            =
            set_reference(data(read_data(pm_phy, paging_data_type(1))
                                        (xlat_idx(1, d‴‘2, addr′))(s2)),
                          Write)))
        ⊃
        unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                    singleton(expr_2_super(translate(1,
                                                                     d‴‘2,
                                                                     addr′,
                                                                     access′,
                                                                     segment_to_priv(d′)))),
                                    ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx
{-3}  xlat_idx(1, d‴‘2, addr′)‘type_of = Mem_ ∧
       xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4}  aligned?(12)(offset(d‴‘2))
{-5}  ∀ (s: (pm_phy‘states)):
        OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
         d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-7}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
               (translate[Physical_memory, pm_phy]
                         (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                   IF leaf
                     THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                         (offset(b))
                   ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                         (offset(b))
                   ENDIF}):
1810      ¬ leaf ⊃
           (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
                      λ (t: [bool, Address]):
                        Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                   (translate[Physical_memory, pm_phy]
```

Rewriting using subset_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

```
linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1:

{-1}  (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), exp
{-2}  unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
                                  singleton(expr_2_super(translate(1,
                                                                   d‴‘2,
                                                                   addr′,
                                                                   access′,
                                                                   segment_to_priv(d′)))),
                                  ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(
{-3}  Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-4}  xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-5}  aligned?(12)(offset(d‴‘2))
{-6}  ∀ (s: (pm_phy‘states)):
        OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
         d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-7}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-8}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
                (translate[Physical_memory, pm_phy]
                          (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
         base:
           {b: Memory_Address_4G |
                    IF leaf
                      THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_leve
                                          (offset(b))
                    ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                  (offset(b))
                    ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-11} pm′‘mem = linear_pm
{-12} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-14} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-15} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-16} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)),
```

Keeping (-2 1) and hiding *,

Case splitting on Transformer_Super_Embedding_Expr [Physical_memory, [bool, Address_Helpers.Address]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (1, d!3‘2, addr!1, access!1, Segment_Types.segment_to_priv(d!1))) = Transformer_Super_Embedding_Expr [Physical_memory, [bool, IA32.Memory_Address_4G]].expr_2_super (Linear_Memory[Physical_memory, pm_phy].translate (1, d!3‘2, addr!1, access!1, Segment_Types.segment_to_priv(d!1))),

we get 2 subgoals:

`linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.1:`

{-1} Transformer_Super_Embedding_Expr[Physical_memory, [bool, Address_Helpers.Address]].expr_2_super
(Linear_Memory[Physical_memory, pm_phy].translate
(1, d'''‘2, addr′, access′, Segment_Types.segment_to_priv(d')))
=
Transformer_Super_Embedding_Expr[Physical_memory, [bool, IA32.Memory_Address_4G]].expr_2_super
(Linear_Memory[Physical_memory, pm_phy].translate
(1, d'''‘2, addr′, access′, Segment_Types.segment_to_priv(d')))

{-2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(translate(1,
d'''‘2,
addr′,
access′,
segment_to_priv(d')))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, ad

---

{1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(translate(1,
d'''‘2,
addr′,
access′,
segment_to_priv(d')))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, ad

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.1`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.2:`

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(translate(1,
$d'''‘2$,
addr′,
access′,
segment_to_priv(d')))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1

{1} Transformer_Super_Embedding_Expr[Physical_memory, [bool, Address_Helpers.Address]].expr_2_super
(Linear_Memory[Physical_memory, pm_phy].translate
(1, $d'''‘2$, addr′, access′, Segment_Types.segment_to_priv(d')))
=
Transformer_Super_Embedding_Expr[Physical_memory, [bool, IA32.Memory_Address_4G]].ex
(Linear_Memory[Physical_memory, pm_phy].translate
(1, $d'''‘2$, addr′, access′, Segment_Types.segment_to_priv(d')))

{2} unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states,
singleton(expr_2_super(translate(1,
$d'''‘2$,
addr′,
access′,
segment_to_priv(d')))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1

Keeping (1) and hiding *,

Applying extensionality,

Expanding the definition(s) of (expr_2_super expr_2_super_res has_next_state),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2:

{-1} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-3} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d'''‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-6} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-15} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]

Hiding formulas: (2 3),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -19 with the terms: (s1!1 s2!1 1 xlat_idx(1, d!3‘2, addr!1)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.1:

{-1} pm′‘states(s'_1)
{-2} pm′‘states(s'_2)
{-3} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-5} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-6} aligned?(12)(offset(d'''‘2))
{-7} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-8} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-9} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-12} pm′‘mem = linear_pm
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-15} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-16} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-17} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2:

{-1} pm′‘states(s_1')
{-2} pm′‘states(s_2')
{-3} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-5} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-6} aligned?(12)(offset(d'''‘2))
{-7} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-8} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-9} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-12} pm′‘mem = linear_pm
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-15} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-16} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-17} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧

Using lemma xlat_pe_in_pt_range,

Instantiating quantified variables,

Case splitting on pm!1‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s1!1))),

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s'_1) ∧
pm′‘states(s'_1) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′,
access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
⊃
pe_in_pt_range?(s'_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))
{-3} pm′‘states(s'_1)
{-4} pm′‘states(s'_2)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-7} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d'''‘2))
{-9} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-10} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) = d''
{-11} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-13} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))

1821

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s'_1) ∧
pm′‘states(s'_1) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′,
access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
⊃
pe_in_pt_range?(s'_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))
{-3} pm′‘states(s'_1)
{-4} pm′‘states(s'_2)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-7} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d'''‘2))
{-9} OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
∧
d''' =
data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
{-10} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) = d''
{-11} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-13} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.2:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s'_1) ∧
pm′‘states(s'_1) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′,
access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
⊃
pe_in_pt_range?(s'_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))
{-3} pm′‘states(s'_1)
{-4} pm′‘states(s'_2)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-7} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d'''‘2))
{-9} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) = d''
{-10} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base: 1825
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.3:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1')))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s_1') ∧
pm′‘states(s_1') ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))‘base_addr, addr′,
access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1'))))‘1
⊃
pe_in_pt_range?(s_1',
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s_1'))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s_1'))))‘2,
addr′))
{-3} pm′‘states(s_1')
{-4} pm′‘states(s_2')
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-7} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d'''‘2))
{-9} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-10} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1')) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1')) = d''
{-11} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-13} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))

1827

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.2:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s'_1) ∧
pm′‘states(s'_1) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′,
access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
⊃
pe_in_pt_range?(s'_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))
{-2} pm′‘states(s'_1)
{-3} pm′‘states(s'_2)
{-4} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-5} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-6} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-7} aligned?(12)(offset(d'''‘2))
{-8} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-9} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) = d''
{-10} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))1829
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

Hiding formulas: -1,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.3:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s'_1) ∧
pm′‘states(s'_1) ∧
OK?(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr, addr′,
access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1))))‘1
⊃
pe_in_pt_range?(s'_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))‘base_addr,
addr′, access′, segment_to_priv(d'))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s'_1))))‘2,
addr′))

{-2} pm′‘states(s'_1)

{-3} pm′‘states(s'_2)

{-4} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))

{-5} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)

{-6} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset

{-7} aligned?(12)(offset(d'''‘2))

{-8} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))

{-9} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s'_1)) = d''

{-10} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))1831

{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.2.2.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3:

{-1} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d‴‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]

Hiding formulas: (3 2),

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -17 with the terms: (s!2 1 xlat_idx(1, d!3‘2, addr!1)),

we get 2 subgoals:

`linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.1:`

{-1} pm′‘states(s'')
{-2} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-4} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-5} aligned?(12)(offset(d'''‘2))
{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-7} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-11} pm′‘mem = linear_pm
{-12} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-14} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-15} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-16} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,

which is trivially true.

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.1`.

`linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2:`

{-1} pm′‘states(s'')
{-2} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-4} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-5} aligned?(12)(offset(d'''‘2))
{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-7} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-11} pm′‘mem = linear_pm
{-12} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-14} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-15} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-16} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,

Simplifying, rewriting, and recording with decision procedures,

Using lemma xlat_idx_memory_address2,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of pe_in_pt_range?,

Simplifying, rewriting, and recording with decision procedures,

Using lemma xlat_pe_in_pt_range,

Instantiating quantified variables,

Case splitting on pm!1‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s!2))),

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s″) ∧
pm′‘states(s″) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))‘1
⊃
pe_in_pt_range?(s″,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))‘base_addr,
addr′, access′, seg-
ment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))))‘2,
addr′))
{-3} 0 ≤ offset(xlat_idx(1, d‴‘2, addr′))
{-4} pm′‘states(s″)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-7} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d‴‘2))
{-9} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-10} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) = d″
{-11} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))1839
{-13} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s″) ∧
pm′‘states(s″) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))‘1
⊃
pe_in_pt_range?(s″,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))))‘2,
addr′))
{-3} 0 ≤ offset(xlat_idx(1, d‴‘2, addr′))
{-4} pm′‘states(s″)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-7} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d‴‘2))
{-9} OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
d‴ =
data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
{-10} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) = d″
{-11} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-13} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.2:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s″) ∧
pm′‘states(s″) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))‘1
⊃
pe_in_pt_range?(s″,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))‘base_addr,
addr′, access′, seg-
ment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))))‘2,
addr′))
{-3} 0 ≤ offset(xlat_idx(1, d‴‘2, addr′))
{-4} pm′‘states(s″)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-7} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d‴‘2))
{-9} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) = d″
{-10} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G | 1843
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.3:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s″) ∧
pm′‘states(s″) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))‘1
⊃
pe_in_pt_range?(s″,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))‘base_addr,
addr′, access′, seg-
ment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))))‘2,
addr′))
{-3} 0 ≤ offset(xlat_idx(1, d‴‘2, addr′))
{-4} pm′‘states(s″)
{-5} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-6} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-7} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-8} aligned?(12)(offset(d‴‘2))
{-9} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-10} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) = d″
{-11} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))1845
{-13} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.2:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s″) ∧
pm′‘states(s″) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))‘1
⊃
pe_in_pt_range?(s″,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))))‘2,
addr′))
{-2} 0 ≤ offset(xlat_idx(1, d‴‘2, addr′))
{-3} pm′‘states(s″)
{-4} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-5} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-6} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-7} aligned?(12)(offset(d‴‘2))
{-8} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-9} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) = d″
{-10} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x₁: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x₁))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool, 1847
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.3:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s″) ∧
pm′‘states(s″) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s″))))‘1
⊃
pe_in_pt_range?(s″,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s″))))‘2,
addr′))
{-2} 0 ≤ offset(xlat_idx(1, d‴‘2, addr′))
{-3} pm′‘states(s″)
{-4} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-5} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-6} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-7} aligned?(12)(offset(d‴‘2))
{-8} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-9} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s″)) = d″
{-10} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x₁: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x₁))
{-12} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool, 1849
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.3.2.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4:

```
{-1}  (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-2}  Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3}  xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4}  aligned?(12)(offset(d‴‘2))
{-5}  ∀ (s: (pm_phy‘states)):
        OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
         d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6}  ∀ (s: (pm_phy‘states)):
        OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-7}  ∀ (s: (pm′‘states)):
        OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
         d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
        ∀ (x1: Physical_memory):
          every[Physical_memory, [bool, Address]]
               (λ (x: Physical_memory): TRUE,
                  λ (t: [bool, Address]):
                    Mem?(t‘2‘type_of) ∧
                     (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                      IF t‘1
                        THEN aligned?(22)(offset(t‘2))
                      ELSE aligned?(12)(offset(t‘2))
                      ENDIF)
                (translate[Physical_memory, pm_phy]
                          (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                        IF leaf
                          THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
                                              (offset(b))
                        ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                     (offset(b))
                        ENDIF}):
        ¬ leaf ⊃
         (∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                    λ (t: [bool, Address]):
                      Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
                 (translate[Physical_memory, pm_phy]
                           (1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
         data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm′‘states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm′‘states)):
        pe_in_pdir_range?(s1,
                          restrict[Address, Memory_Address_4G, boolean]
                                  ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         =
         pe_in_pdir_range?(s2,
                           restrict[Address, Memory_Address_4G, boolean]
                                   ((pm′‘ro_addr ∪ pm′‘rw_addr)))
         ∧
         pe_in_ptab_range?(s1,
                           restrict[Address, Memory_Address_4G, boolean]
```

Hiding formulas: 2, 3,

Using lemma subset_ptab_address_rw_addr,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1:

{-1} (address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, bool](pm′`ro_addr) ∪ restrict [Address,
∧
(address_block(xlat_idx(1, d'''`2, addr′), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, boo
⊃ (address_block(xlat_idx(1, d'''`2, addr′), expt(2, pe_size)) ⊆ pm_phy`rw_addr)

{-2} (addresses ⊆ ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(1, d'''`2, addr′), expt(2, 2))))

{-3} Mem?(xlat_idx(1, d'''`2, addr′)`type_of)

{-4} xlat_idx(1, d'''`2, addr′)`offset < max_linear_offset

{-5} aligned?(12)(offset(d'''`2))

{-6} ∀ (s: (pm_phy`states)):
OK?(translate(0, d''`base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''`base_addr, addr′, access′, segment_to_priv(d'))(s))

{-7} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''

{-8} ∀ (s: (pm′`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr′, access′, segment_to_priv(cs))(x_1))

{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))

{-11} pm′`mem = linear_pm

{-12} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-13} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))

{-14} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-15} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))

{-16} ∀ (s_1, s_2: (pm′`states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]

Simplifying, rewriting, and recording with decision procedures,

Applying propositional simplification,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1:

{-1} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d‴‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]

Using lemma pm_address_in_pt_range_in_rw_addr,

Case splitting on restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)),

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.1:

{-1} restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)) =
(restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Address, Memory_Address_4G,
{-2} is_linear_plain_memory?(pm′) ∧ pm′`states(s′) ⊃
(address_in_pt_range?(s′, restrict [Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr))
{-3} (addresses ⊆ ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(1, d‴`2, addr′), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d‴`2, addr′)`type_of)
{-5} xlat_idx(1, d‴`2, addr′)`offset < max_linear_offset
{-6} aligned?(12)(offset(d‴`2))
{-7} ∀ (s: (pm_phy`states)):
OK?(translate(0, d″`base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″`base_addr, addr′, access′, segment_to_priv(d′))(s))
{-8} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-9} ∀ (s: (pm′`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-12} pm′`mem = linear_pm
{-13} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-14} ∀ (s1, s2: (pm′`states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-15} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-16} ∀ (s1, s2: (pm′`states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-17} ∀ (s1, s2: (pm′`states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.1`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.2:

{-1} is_linear_plain_memory?(pm′) ∧ pm′‘states(s′) ⊃
(address_in_pt_range?(s′, restrict [Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-4} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-5} aligned?(12)(offset(d‴‘2))
{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-7} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-11} pm′‘mem = linear_pm
{-12} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-14} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-15} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-16} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2:

{-1} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d‴‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1:

{-1} pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, bool]
((pm′`ro_addr ∪ pm′`rw_addr)),
1)
(xlat_idx(1, d‴`2, addr′))
⊃
(address_block(xlat_idx(1, d‴`2, addr′), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, boo
{-2} (addresses ⊆ ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(1, d‴`2, addr′), expt(2, 2))))
{-3} Mem?(xlat_idx(1, d‴`2, addr′)`type_of)
{-4} xlat_idx(1, d‴`2, addr′)`offset < max_linear_offset
{-5} aligned?(12)(offset(d‴`2))
{-6} ∀ (s: (pm_phy`states)):
OK?(translate(0, d″`base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″`base_addr, addr′, access′, segment_to_priv(d′))(s))
{-7} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-8} ∀ (s: (pm′`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-11} pm′`mem = linear_pm
{-12} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-13} ∀ (s1, s2: (pm′`states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-14} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-15} ∀ (s1, s2: (pm′`states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-16} ∀ (s1, s2: (pm′`states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma xlat_pe_in_pt_range,

Instantiating quantified variables,

Case splitting on pm!1‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s!1))),

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-3} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-5} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-6} aligned?(12)(offset(d‴‘2))
{-7} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-8} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-9} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base: 1865
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-3} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-5} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-6} aligned?(12)(offset(d‴‘2))
{-7} OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
d‴ =
data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
{-8} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-9} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.2:

  {-1}  pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
  {-2}  union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
          is_linear_plain_memory?(pm′) ∧
           pm′‘states(s′) ∧
            pm′‘states(s′) ∧
             OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
                           addr′, access′, segment_to_priv(d′))
                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
              ∧
              ¬ data(translate(pdir_lvl,
                               data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
                               addr′, access′, segment_to_priv(d′))
                              (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
          ⊃
          pe_in_pt_range?(s′,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                          ptab_lvl)
                         (xlat_idx(ptab_lvl,
                                   data(translate(pdir_lvl,
                                                  data(read_data(pm_phy, pdbr_data_type)
                                                               (PDBR)
                                                               (s′))‘base_addr,
                                                  addr′, access′, seg-
        ment_to_priv(d′))
                                                 (state(read_data(pm_phy, pdbr_data_type)
                                                               (PDBR)
                                                               (s′))))‘2,
                                              addr′))
  {-3}  (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d′′′‘2, addr′), expt(2, 2))))
  {-4}  Mem?(xlat_idx(1, d′′′‘2, addr′)‘type_of)
  {-5}  xlat_idx(1, d′′′‘2, addr′)‘offset < max_linear_offset
  {-6}  aligned?(12)(offset(d′′′‘2))
  {-7}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d′′
  {-8}  ∀ (s: (pm′‘states)):
          OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
           d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
  {-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
          ∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                   λ (t: [bool, Address]):
                     Mem?(t‘2‘type_of) ∧
                      (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                       IF t‘1
                          THEN aligned?(22)(offset(t‘2))
                       ELSE aligned?(12)(offset(t‘2))
                       ENDIF)
                (translate[Physical_memory, pm_phy]
                      (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
 {-10}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                   IF leaf
                      THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] 1869 bits_per_level [Physical_
                                          (offset(b))
                   ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                 (offset(b))
                   ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.3:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-3} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-4} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-5} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-6} aligned?(12)(offset(d‴‘2))
{-7} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-8} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-9} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-11} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base: 1871
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.2:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))

{-2} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))

{-3} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)

{-4} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset

{-5} aligned?(12)(offset(d‴‘2))

{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))

{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″

{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))

{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G | 1873
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃

Hiding formulas: -1,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Using lemma unchanged_memory_invariant_invariant,

Using lemma pm_states,

Using lemma expr_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.3:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))

{-2} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))

{-3} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)

{-4} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset

{-5} aligned?(12)(offset(d‴‘2))

{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))

{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″

{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))

{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G | 1875
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.2:

{-1} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d‴‘2, addr′), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d‴‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d″
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-15} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.2:

{-1} (addresses ⊆ ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2))))
{-2} Mem?(xlat_idx(1, d'''‘2, addr′)‘type_of)
{-3} xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d'''‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))
{-6} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-10} pm′‘mem = linear_pm
{-11} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-12} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-13} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-14} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-15} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.1.4.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2:

{-1} ((address_block(xlat_idx(1, d'''`2, addr$'$), expt(2, 2)) ⊆ pm_phy`rw_addr) ∧
(∀ (s: (pm$'$`states)):
OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''`2, addr$'$))(s)))
∧
(∀ (s_1, s_2: (pm$'$`states)):
set_reference(data(read_data(pm_phy, paging_data_type(1))
(xlat_idx(1, d'''`2, addr$'$))(s_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(1))
(xlat_idx(1, d'''`2, addr$'$))(s_2)),
Write)))
⊃
unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(translate(1,
d'''`2,
addr$'$,
access$'$,
segment_to_priv(d')))),
((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(1, d'''`2,
{-2} xlat_idx(1, d'''`2, addr$'$)`type_of = Mem_ ∧
xlat_idx(1, d'''`2, addr$'$)`offset < max_linear_offset
{-3} aligned?(12)(offset(d'''`2))
{-4} ∀ (s: (pm_phy`states)):
OK?(translate(0, d''`base_addr, addr$'$, access$'$, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''`base_addr, addr$'$, access$'$, segment_to_priv(d'))(s))
{-5} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-6} ∀ (s: (pm$'$`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr$'$, access$'$, segment_to_priv(cs))(x_1))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr$'$, access$'$, segment_to_priv(cs))(x_1)))
{-9} ∀ (

Hiding formulas: (2 3),

Replacing using formula -24,

Rewriting using subset_of_differences, matching in *,

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1:

{-1} pe_in_pt_range?(s', restrict[Address, Memory_Address_4G, boolean] ((pm′‘ro_addr ∪ pm′‘rw_addr)), 1) (xlat_idx(1, d'''‘2, addr′)) ⊃ (address_block(xlat_idx(1, d'''‘2, addr′), expt(2, pe_size)) ⊆ extend [Address, Memory_Address_4G, boo

{-2} ((address_block(xlat_idx(1, d'''‘2, addr′), expt(2, 2)) ⊆ pm_phy‘rw_addr) ∧
(∀ (s: (pm′‘states)): OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''‘2, addr′))(s)))
∧
(∀ (s_1, s_2: (pm′‘states)): set_reference(data(read_data(pm_phy, paging_data_type(1)) (xlat_idx(1, d'''‘2, addr′))(s_1)), Write) = set_reference(data(read_data(pm_phy, paging_data_type(1)) (xlat_idx(1, d'''‘2, addr′))(s_2)), Write)))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm_phy‘states, singleton(expr_2_super(translate(1, d'''‘2, addr′, access′, segment_to_priv(d')))), ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ address_block(xlat_idx(1, d'''‘2,

{-3} xlat_idx(1, d'''‘2, addr′)‘type_of = Mem_ ∧
xlat_idx(1, d'''‘2, addr′)‘offset < max_linear_offset

{-4} aligned?(12)(offset(d'''‘2))

{-5} ∀ (s: (pm_phy‘states)): OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧ d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))

{-6} ∀ (s: (pm_phy‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧ data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''

{-7} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧ d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type): ∀ (x_1: Physical_memory): every[Physical_memory, [bool, Address]] (λ (x: Physical_memory): TRUE, λ (t: [bool, Address]): Mem?(t‘2‘type_of) ∧ (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧ IF t‘1 THEN aligned?(22)(offset(t‘2)) ELSE aligned?(12)(offset(t‘2)) ENDIF) (translate[Physical_memory, pm_phy] (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))

{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool, base: {b: Memory_Address_4G | IF leaf THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical (offset(b)) ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size) (offset(b)) ENDIF}):
¬ leaf ⊃

Hiding formulas: -2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma xlat_pe_in_pt_range,

Instantiating quantified variables,

Case splitting on pm!1‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s!1))),

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-3} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-4} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-5} aligned?(12)(offset(d‴‘2))
{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G | 1885
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃

Instantiating quantified variables,

we get 3 subgoals:

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.1:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-3} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-4} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-5} aligned?(12)(offset(d‴‘2))
{-6} OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
d‴ =
data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))1887
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.1`.

```
linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.2:

  {-1}  pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
  {-2}  union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
          is_linear_plain_memory?(pm′) ∧
           pm′‘states(s′) ∧
            pm′‘states(s′) ∧
             OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
                           addr′, access′, segment_to_priv(d′))
                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
              ∧
              ¬ data(translate(pdir_lvl,
                               data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
                               addr′, access′, segment_to_priv(d′))
                              (state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
          ⊃
          pe_in_pt_range?(s′,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                          ptab_lvl)
                         (xlat_idx(ptab_lvl,
                                   data(translate(pdir_lvl,
                                                  data(read_data(pm_phy, pdbr_data_type)
                                                               (PDBR)
                                                               (s′))‘base_addr,
                                                  addr′, access′, seg-
        ment_to_priv(d′))
                                                 (state(read_data(pm_phy, pdbr_data_type)
                                                               (PDBR)
                                                               (s′))))‘2,
                                   addr′))
  {-3}  Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
  {-4}  xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
  {-5}  aligned?(12)(offset(d‴‘2))
  {-6}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
  {-7}  ∀ (s: (pm′‘states)):
          OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
           d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
  {-8}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
          ∀ (x1: Physical_memory):
            every[Physical_memory, [bool, Address]]
                 (λ (x: Physical_memory): TRUE,
                   λ (t: [bool, Address]):
                     Mem?(t‘2‘type_of) ∧
                      (0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
                       IF t‘1
                         THEN aligned?(22)(offset(t‘2))
                       ELSE aligned?(12)(offset(t‘2))
                       ENDIF)
                (translate[Physical_memory, pm_phy]
                      (0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
  {-9}  ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
           base:
             {b: Memory_Address_4G |
                   IF leaf
                     THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
                                         (offset(b))                                          1889
                   ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
                                 (offset(b))
                   ENDIF}):
          ¬ leaf ⊃
           (∀ (x1: Physical_memory):
              every[Physical_memory, [bool, Address]]
                   (λ (x: Physical_memory): TRUE,
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.3:

{-1} pm′‘states(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-3} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-4} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-5} aligned?(12)(offset(d‴‘2))
{-6} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-8} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-10} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G | 1891
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.2:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-2} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d‴‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf 1893
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):

Hiding formulas: -1,

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.3:

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘states(s′) ∧
OK?(translate(pdir_lvl, data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))
∧
¬ data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)(s′))))‘1
⊃
pe_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
ptab_lvl)
(xlat_idx(ptab_lvl,
data(translate(pdir_lvl,
data(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))‘base_addr,
addr′, access′, segment_to_priv(d′))
(state(read_data(pm_phy, pdbr_data_type)
(PDBR)
(s′))))‘2,
addr′))
{-2} Mem?(xlat_idx(1, d‴‘2, addr′)‘type_of)
{-3} xlat_idx(1, d‴‘2, addr′)‘offset < max_linear_offset
{-4} aligned?(12)(offset(d‴‘2))
{-5} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s)) ∧
d‴ = data(translate(0, d″‘base_addr, addr′, access′, segment_to_priv(d′))(s))
{-6} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s′)) = d″
{-7} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x1))
{-9} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x1: Physical_memory):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.1.3`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.2:

{-1} ((address_block(xlat_idx(1, d'''`2, addr$'$), expt(2, 2)) ⊆ pm_phy`rw_addr) ∧
(∀ (s: (pm$'$`states)):
OK?(read_data(pm_phy, paging_data_type(1))(xlat_idx(1, d'''`2, addr$'$))(s)))
∧
(∀ (s_1, s_2: (pm$'$`states)):
set_reference(data(read_data(pm_phy, paging_data_type(1))
(xlat_idx(1, d'''`2, addr$'$))(s_1)),
Write)
=
set_reference(data(read_data(pm_phy, paging_data_type(1))
(xlat_idx(1, d'''`2, addr$'$))(s_2)),
Write)))
⊃
unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
singleton(expr_2_super(translate(1,
d'''`2,
addr$'$,
access$'$,
segment_to_priv(d')))),
((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ address_block(xlat_idx(1, d'''`2,
{-2} xlat_idx(1, d'''`2, addr$'$)`type_of = Mem_ ∧
xlat_idx(1, d'''`2, addr$'$)`offset < max_linear_offset
{-3} aligned?(12)(offset(d'''`2))
{-4} ∀ (s: (pm_phy`states)):
OK?(translate(0, d''`base_addr, addr$'$, access$'$, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''`base_addr, addr$'$, access$'$, segment_to_priv(d'))(s))
{-5} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''
{-6} ∀ (s: (pm$'$`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-7} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr$'$, access$'$, segment_to_priv(cs))(x_1))
{-8} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical_
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr$'$, access$'$, segment_to_priv(cs))(x_1)))
{-9} ∀ (

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.1.2.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.3.2:

{-1} ∀ (s: (pm_phy‘states)):
OK?(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s)) ∧
d''' = data(translate(0, d''‘base_addr, addr′, access′, segment_to_priv(d'))(s))

{-2} ∀ (s: (pm_phy‘states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''

{-3} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-4} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))

{-5} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))

{-6} pm′‘mem = linear_pm

{-7} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-8} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))

{-9} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-10} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))

{-11} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]

Instantiating quantified variables,

Using lemma translate_result_no_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.3.2`.

linear_resolve_unchanged_pm_phy.1.1.3.3.4:

{-1} ∀ (s: (pm_phy`states)):
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) ∧
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s)) = d''

{-2} ∀ (s: (pm′`states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d' = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-3} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧
(0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset) ∧
IF t`1
THEN aligned?(22)(offset(t`2))
ELSE aligned?(12)(offset(t`2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr`base_addr, addr′, access′, segment_to_priv(cs))(x_1))

{-4} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))

{-5} pm′`mem = linear_pm

{-6} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-7} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))

{-8} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-9} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))

{-10} ∀ (s_1, s_2: (pm′`states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))

{-11} ∀ (s: (pm′`states), lvl: Level,
a:

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -8 with the terms: d!2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.3.4`.

linear_resolve_unchanged_pm_phy.1.1.3.4:

{-1} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-2} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))
{-3} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-4} pm′‘mem = linear_pm
{-5} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-6} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-7} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-8} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-9} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-10} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),

Keeping (-3 1) and hiding *,

Instantiating quantified variables,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.4`.

linear_resolve_unchanged_pm_phy.1.1.3.5:

{-1} ∀ (s: (pm′‘states)):
OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s)) ∧
d′ = data(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-2} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type):
∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧
(0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset) ∧
IF t‘1
THEN aligned?(22)(offset(t‘2))
ELSE aligned?(12)(offset(t‘2))
ENDIF)
(translate[Physical_memory, pm_phy]
(0, pdbr‘base_addr, addr′, access′, segment_to_priv(cs))(x_1))

{-3} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t‘2‘type_of) ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))

{-4} pm′‘mem = linear_pm

{-5} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))

{-6} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))

{-7} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))

{-8} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))

{-9} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))

{-10} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),

Keeping (-2 1) and hiding *,

Instantiating quantified variables,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.1.3.5`.

linear_resolve_unchanged_pm_phy.1.2:

{-1} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))
ELSE aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(b))
ENDIF}):
¬ leaf ⊃
(∀ (x_1: Physical_memory):
every[Physical_memory, [bool, Address]]
(λ (x: Physical_memory): TRUE,
λ (t: [bool, Address]):
Mem?(t`2`type_of) ∧ 0 ≤ t`2`offset ∧ t`2`offset < max_linear_offset)
(translate[Physical_memory, pm_phy]
(1, base, addr′, access′, segment_to_priv(cs))(x_1)))
{-2} pm′`mem = linear_pm
{-3} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-4} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-5} ∀ (s: (pm′`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-6} ∀ (s_1, s_2: (pm′`states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-7} ∀ (s_1, s_2: (pm′`states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
{-8} ∀ (s: (pm′`states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-9} ∀ (s_1, s_2: (pm′`states), lvl: Level,
a:
(pe_in_pt_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) 1907
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-10} (pm′`other_actions ⊆ pm_phy`other_actions)
{-11} (linear_resolve_register_transformers ⊆ pm_phy`other_actions)
{-12} ∀ (a: ((pm′`ro_addr ∪ pm′`rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-13} ∀ (s: (pm′`states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr),

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Using lemma translate_memory_address,

Using lemma translate_result_leaf,

Using lemma translate_result_no_leaf,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.2`.

linear_resolve_unchanged_pm_phy.1.3:

{-1} pm′‘mem = linear_pm
{-2} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-4} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-6} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-7} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-8} ∀ (s1, s2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-9} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-10} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-11} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-12} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-13} pm′‘states = pm_phy‘states
{-14} plain_memory?(pm_phy)
{-15} (addresses ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-16} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](address_in_pt
= addresses
{-17} Mem?(addr′‘type_of)
{-18} 0 ≤ addr′‘offset
{-19} addr′‘offset < max_linear_offset
{-20} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-21} is_linear_plain_memory?(pm′)
{-22} pm_phy‘states(s′)

{1} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, leaf: bool,
base:
{b: Memory_Address_4G |
IF leaf
THEN aligned?(−1 × (pdir_lvl [Physical_memory, pm_phy] × bits_per_level [Physical
(offset(b))

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Expanding the definition of every,

Using lemma translate_memory_address,

Using lemma translate_result_leaf,

Using lemma translate_result_no_leaf,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.3`.

`linear_resolve_unchanged_pm_phy.1.4:`

{-1} pm′‘mem = linear_pm
{-2} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-4} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-6} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-7} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-8} ∀ (s_1, s_2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-9} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-10} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-11} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-12} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-13} pm′‘states = pm_phy‘states
{-14} plain_memory?(pm_phy)
{-15} (addresses ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-16} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](address_in_pt
= addresses
{-17} Mem?(addr′‘type_of)
{-18} 0 ≤ addr′‘offset
{-19} addr′‘offset < max_linear_offset
{-20} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-21} is_linear_plain_memory?(pm′)
{-22} pm_phy‘states(s′)

{1} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, x_1: Physical_memory, t: [bool, Address]):
¬ t‘1 ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset ∧ Mem?(t‘2‘type_of) ⊃
offset(t‘2) ≥ 0
{2} unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm_phy‘states)

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_unchanged_pm_phy.1.4`.

`linear_resolve_unchanged_pm_phy.1.5:`

{-1} pm′‘mem = linear_pm
{-2} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-3} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s2))
{-4} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-5} ∀ (s1, s2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s2))
{-6} ∀ (s1, s2: (pm′‘states)):
pe_in_pdir_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-7} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-8} ∀ (s1, s2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s2)), Write)
{-9} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-10} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-11} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-12} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-13} pm′‘states = pm_phy‘states
{-14} plain_memory?(pm_phy)
{-15} (addresses ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-16} ((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](address_in_pt
= addresses
{-17} Mem?(addr′‘type_of)
{-18} 0 ≤ addr′‘offset
{-19} addr′‘offset < max_linear_offset
{-20} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-21} is_linear_plain_memory?(pm′)
{-22} pm_phy‘states(s′)

{1} ∀ (cs: Segment_Reg_type, pdbr: Pdbr_type, x1: Physical_memory, t: [bool, Address]):
t‘1 ∧ 0 ≤ t‘2‘offset ∧ t‘2‘offset < max_linear_offset ∧ Mem?(t‘2‘type_of) ⊃
offset(t‘2) ≥ 0
{2} unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm_phy‘states)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_unchanged_pm_phy.1.5`.
`linear_resolve_unchanged_pm_phy.2`:

{-1}	((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Address_4G, bool, FALSE](add = addresses
{-2}	Mem?(addr′‘type_of)
{-3}	0 ≤ addr′‘offset
{-4}	addr′‘offset < max_linear_offset
{-5}	union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-6}	is_linear_plain_memory?(pm′)
{-7}	pm′‘states(s′)
{1}	(addresses ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2}	unchanged_memory_invariant?[Physical_memory] (pm_phy‘mem, pm′‘states, singleton(expr_2_super[Physical_memory, Address](linear_resolve(addr′, access′))), addresses)

Hiding formulas: 2,
Replacing using formula -1,
Rewriting using difference_subset, matching in *,
This completes the proof of `linear_resolve_unchanged_pm_phy.2`.
Q.E.D.

C.117.50 Linear_Memory_Properties.linear_resolve_read_transformers_ok

Terse proof for `linear_resolve_read_transformers_ok`.
`linear_resolve_read_transformers_ok`:

{1}	∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G): is_linear_plain_memory?(pm) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr) ⊃ transformers_ok?(pm‘states, singleton(expr_2_super(linear_resolve(addr, Read))))

Installing automatic rewrites from: (singleton ok_expr_2_super)
Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -7,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_linear_resolve_read_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_read_transformers_ok`.
Q.E.D.

C.117.51 Linear_Memory_Properties.linear_resolve_write_transformers_ok

Terse proof for `linear_resolve_write_transformers_ok`.

`linear_resolve_write_transformers_ok`:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G):
is_linear_plain_memory?(pm) ∧ pm`rw_addr(addr) ⊃
transformers_ok?(pm`states, singleton(expr_2_super(linear_resolve(addr, Write))))

Installing automatic rewrites from: ok_expr_2_super
Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -7,
Using lemma pm_linear_resolve_write_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_write_transformers_ok`.
Q.E.D.

C.117.52 Linear_Memory_Properties.linear_resolve_transformer_invariant

Terse proof for `linear_resolve_transformer_invariant`.

`linear_resolve_transformer_invariant`:

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, access: Memory_access):
is_linear_plain_memory?(pm) ∧ union(pm`ro_addr, pm`rw_addr)(addr) ⊃
transformer_invariant?(pm`states, singleton(expr_2_super(linear_resolve(addr, access))))

Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Using lemma linear_resolve_unchanged_pm_phy,
Using lemma unchanged_memory_invariant_invariant[Physical_memory],
Using lemma pm_states,
Expanding the definition of transformer_invariant?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-9 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
Applying decompose-equality,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_transformer_invariant`.
Q.E.D.

C.117.53 Linear_Memory_Properties.linear_resolve_states_TCC1

Terse proof for `linear_resolve_states_TCC1`.

`linear_resolve_states_TCC1`:

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          addr: Memory_Address_4G, access: Memory_access):
       union(pm`ro_addr, pm`rw_addr)(addr) ∧
        is_linear_plain_memory?(pm) ∧ pm`states(s) ∧ OK?(linear_resolve(addr, access)(s))
        ⊃
        OK?[Physical_memory, Address]
             (linear_resolve[Physical_memory, pm_phy](addr, access)(s))
         ∨
         Exception?[Physical_memory, Address]
              (linear_resolve[Physical_memory, pm_phy](addr, access)(s))
```

Repeatedly Skolemizing and flattening,
This completes the proof of `linear_resolve_states_TCC1`.
Q.E.D.

C.117.54 Linear_Memory_Properties.linear_resolve_states

Terse proof for `linear_resolve_states`.

`linear_resolve_states`:

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          addr: Memory_Address_4G, access: Memory_access):
       union(pm`ro_addr, pm`rw_addr)(addr) ∧
        is_linear_plain_memory?(pm) ∧ pm`states(s) ∧ OK?(linear_resolve(addr, access)(s))
        ⊃ pm_phy`states(state(linear_resolve(addr, access)(s)))
```

Installing automatic rewrites from: has_next_state
Repeatedly Skolemizing and flattening,
Using lemma linear_resolve_unchanged_pm_phy,
Using lemma pm_states,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -2,
Using lemma unchanged_memory_invariant_invariant,
Using lemma super_transformer_invariant_next_ok,
Installing automatic rewrites from: (expr_2_super! expr_2_super_res!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of singleton,
which is trivially true.
This completes the proof of `linear_resolve_states`.
Q.E.D.

C.117.55 Linear_Memory_Properties.linear_resolve_same_result

Terse proof for `linear_resolve_same_result`.

`linear_resolve_same_result:`

{1} ∀ (pm: Plain_Memory[Linear_memory], addr: Memory_Address_4G, ac1, ac2: Memory_access,
s_1, s_2: Linear_memory):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
is_linear_plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
OK?(linear_resolve(addr, ac1)(s_1)) ∧ OK?(linear_resolve(addr, ac2)(s_2))
⊃ data(linear_resolve(addr, ac1)(s_1)) = data(linear_resolve(addr, ac2)(s_2))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: has_next_state ## ok_result singleton max_level bus_width bits_per_level Mem min_linear Address_Helpers.<= ptab_lvl pdir_lvl max_linear < pe_size

Case splitting on interpreted_data_type?(pdbr_data_type), interpreted_data_type?(segment_reg_data_type),

we get 3 subgoals:

`linear_resolve_same_result.1:`

{-1} interpreted_data_type?(segment_reg_data_type)
{-2} interpreted_data_type?(pdbr_data_type)
{-3} Mem?(addr′‘type_of)
{-4} 0 ≤ addr′‘offset
{-5} addr′‘offset < max_linear_offset
{-6} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s_1')
{-9} pm′‘states(s_2')
{-10} OK?(linear_resolve(addr′, ac1′)(s_1'))
{-11} OK?(linear_resolve(addr′, ac2′)(s_2'))

{1} data(linear_resolve(addr′, ac1′)(s_1')) = data(linear_resolve(addr′, ac2′)(s_2'))

Case splitting on OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1)) AND OK?(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s1!1)))) AND OK?(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))))),

we get 5 subgoals:

`linear_resolve_same_result.1.1:`

{-1}	OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) ∧ OK?(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))) ∧ OK?(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s_1'))))‘base_addr, addr′, ac1′, segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS) (s_1')))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s_1'))))))
{-2}	interpreted_data_type?(segment_reg_data_type)
{-3}	interpreted_data_type?(pdbr_data_type)
{-4}	Mem?(addr′‘type_of)
{-5}	0 ≤ addr′‘offset
{-6}	addr′‘offset < max_linear_offset
{-7}	union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-8}	is_linear_plain_memory?(pm′)
{-9}	pm′‘states(s_1')
{-10}	pm′‘states(s_2')
{-11}	OK?(linear_resolve(addr′, ac1′)(s_1'))
{-12}	OK?(linear_resolve(addr′, ac2′)(s_2'))
{1}	data(linear_resolve(addr′, ac1′)(s_1')) = data(linear_resolve(addr′, ac2′)(s_2'))

Case splitting on OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s2!1)) AND OK?(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s2!1)))) AND OK?(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s2!1))))‘base_addr, addr!1, ac2!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))))),

we get 5 subgoals:

`linear_resolve_same_result.1.1.1:`

{-1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')) ∧
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))

{-2} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) ∧
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))

{-3} interpreted_data_type?(segment_reg_data_type)
{-4} interpreted_data_type?(pdbr_data_type)
{-5} Mem?(addr′‘type_of)
{-6} 0 ≤ addr′‘offset
{-7} addr′‘offset < max_linear_offset
{-8} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-9} is_linear_plain_memory?(pm′)
{-10} pm′‘states(s_1')
{-11} pm′‘states(s_2')
{-12} OK?(linear_resolve(addr′, ac1′)(s_1'))
{-13} OK?(linear_resolve(addr′, ac2′)(s_2'))

{1} data(linear_resolve(addr′, ac1′)(s_1')) = data(linear_resolve(addr′, ac2′)(s_2'))

Applying disjunctive simplification to flatten sequent,
Hiding formulas: (-16 -17),
Using lemma translate_memory_address,
Using lemma translate_memory_address,
Case splitting on data(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1)) = data(read_data(pm_phy,

segment_reg_data_type)(CS)(s2!1)), pm!1‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1))),
pm!1‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s2!1))),

we get 4 subgoals:

linear_resolve_same_result.1.1.1.1:

{-1} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-2} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-4} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear
{-5} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,

Case splitting on data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s1!1))))‘base_addr = data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type)(CS) (s2!1))))‘base_addr, pm!1‘states (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type) (CS) (s1!1))))), pm!1‘states (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data(pm_phy, segment_reg_data_type) (CS) (s2!1))))),

we get 4 subgoals:

linear_resolve_same_result.1.1.1.1.1:

{-1} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))
{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))
{-3} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))‘base_addr
{-4} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-5} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-7} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear
{-8} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)

Case splitting on data(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s2!1))))‘base_addr, addr!1, ac2!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))))) = data(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))))),

we get 4 subgoals:

linear_resolve_same_result.1.1.1.1.1.1:

{-1} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))`base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
=
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))

{-2} pm′`states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))

{-3} pm′`states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))

{-4} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))`base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))`base_addr

{-5} pm′`states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))

{-6} pm′`states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))

{-7} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-8} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,

Case splitting on data(translate(0, data(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pdbr_data_type)(PDBR) (state(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))))‘1,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.1:

{-1} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))`1

{-2} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))`base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
=
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))

{-3} pm′`states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))

{-4} pm′`states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))

{-5} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))`base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))`base_addr

{-6} pm′`states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))

{-7} pm′`states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))

{-8} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-9} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -2,

Replacing using formula -1,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.1`.

linear_resolve_same_result.1.1.1.1.1.1.2:

{-1} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))`base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
=
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))

{-2} pm′`states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))

{-3} pm′`states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))

{-4} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))`base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))`base_addr

{-5} pm′`states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))

{-6} pm′`states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))

{-7} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-8} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Replacing using formula 1,

Case splitting on OK?(translate(1, data(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))))‘2, addr!1, ac1!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))))))), OK?(translate(1, data(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))))‘2, addr!1, ac2!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))) (state(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1))))‘base_addr, addr!1, ac2!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))))))),

we get 3 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.2.1:

{-1} OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))))‘2,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s_2'))))))))

{-2} OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))))‘2,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -6,

Replacing using formula -9,

Case splitting on pm!1‘states (state(translate(0, data(read_data(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac1!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))))), pm!1‘states (state(translate(0, data(read_data(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1))))‘base_addr, addr!1, ac2!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s1!1)))) (state(read_data(pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s2!1)))))),

we get 3 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.1:

  {-1}  pm′‘states
          (state(translate(0,
                           data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′1))))‘base_addr,
                           addr′, ac2′,
                           segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                          (s′1))))
                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′2)))))))
  {-2}  pm′‘states
          (state(translate(0,
                           data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′1))))‘base_addr,
                           addr′, ac1′,
                           segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                          (s′1))))
                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′1)))))))
  {-3}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))‘base_addr,
                                     addr′, ac1′,
                                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1)))))‘2,
                      addr′, ac2′,
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                     (s′1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                     (state(read_data(pm_phy, segment_reg_data_type)
                                                                     (CS)
                                                                     (s′1))))‘base_addr,
                                      addr′, ac2′,
                                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                     (CS)
                                                                     (s′1))))
                                     (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                     (state(read_data(pm_phy,
                                                                      segment_reg_data_type)
```

Using lemma translate_same_result,

we get 2 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1:

  {-1}  union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
         is_linear_plain_memory?(pm′) ∧
          pe_in_pt_range?(state(translate(0,
                                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                        (state(read_data
                                                                 (pm_phy, segment_reg_data_type)
                                                                 (CS)
                                                                 (s′1))))‘base_addr,
                                          addr′, ac1′,
                                          segment_to_priv(data(read_data
                                                                 (pm_phy, segment_reg_data_type)
                                                                 (CS)
                                                                 (s′1))))
                                        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                        (state(read_data
                                                                 (pm_phy, segment_reg_data_type)
                                                                 (CS)
                                                                 (s′1))))),
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                          1)
                         (xlat_idx(1,
                                   data(translate(0,
                                                  data(read_data(pm_phy, pdbr_data_type)
                                                                (PDBR)
                                                                (state
                                                                  (read_data
                                                                    (pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))‘base_addr,
                                                  addr′, ac1′,
                                                  segment_to_priv(data
                                                                    (read_data
                                                                      (pm_phy, segment_reg_data_type)
                                                                      (CS)
                                                                      (s′1))))
                                                (state(read_data(pm_phy, pdbr_data_type)
                                                                (PDBR)
                                                                (state
                                                                  (read_data
                                                                    (pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))‘2,
                                                addr′))
           ∧
           OK?(translate(1,
                         data(translate(0,
                                        data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                      (state(read_data
                                                               (pm_phy, segment_reg_data_type)
                                                               (CS)
                                                               (s′1))))‘base_addr,
                                        addr′, ac1′,
                                        segment_to_priv(data(read_data(pm_phy,
                                                                       segment_reg_data_type)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.1:

  {-1}  union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
  {-2}  is_linear_plain_memory?(pm′)
  {-3}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))‘base_addr,
                                     addr′, ac1′,
                                     segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))))‘2,
                      addr′, ac1′,
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                                   (s′1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))‘base_addr,
                                      addr′, ac1′,
                                      segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))
                                     (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy,
                                                                     seg-
      ment_reg_data_type)
                                                                                    (CS)
                                                                                    (s′1)))))))))
  {-4}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))‘base_addr,
                                     addr′, ac1′,
                                     segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))))‘2,
                      addr′, ac2′,
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                                   (s′1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
```

Replacing using formula -5,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.1`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.2:

  {-1}  union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
  {-2}  is_linear_plain_memory?(pm′)
  {-3}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))‘base_addr,
                                     addr′, ac1′,
                                     segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))))‘2,
                      addr′, ac1′,
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                                   (s′1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))‘base_addr,
                                      addr′, ac1′,
                                      segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))
                                     (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy,
                                                                     seg-
      ment_reg_data_type)
                                                                                    (CS)
                                                                                    (s′1)))))))))
  {-4}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))‘base_addr,
                                     addr′, ac1′,
                                     segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
                                                                                   (CS)
                                                                                   (s′1))))))‘2,
                      addr′, ac2′,
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                                   (s′1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, seg-
      ment_reg_data_type)
```

Hiding formulas: (-3 -4 3),

Using lemma xlat_pe_in_pt_range,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.1.1.2`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.1.2:

  {-1}  pm′‘states
          (state(translate(0,
                           data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′1))))‘base_addr,
                           addr′, ac2′,
                           segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                          (s′1))))
                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′2)))))))
  {-2}  pm′‘states
          (state(translate(0,
                           data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′1))))‘base_addr,
                           addr′, ac1′,
                           segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                          (s′1))))
                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s′1)))))))
  {-3}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))‘base_addr,
                                     addr′, ac1′,
                                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s′1))))))‘2,
                      addr′, ac2′,
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                     (s′1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                     (state(read_data(pm_phy, segment_reg_data_type)
                                                                     (CS)
                                                                     (s′1))))‘base_addr,
                                      addr′, ac2′,
                                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                     (CS)
                                                                     (s′1))))
                                     (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                     (state(read_data(pm_phy,
                                                                      segment_reg_data_type)
```

Hiding formulas: (-1 -2 3),

Using lemma translate_result_no_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.1.2`.

linear_resolve_same_result.1.1.1.1.1.1.2.1.2:

{-1} pm′‘states
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))))))

{-2} OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1')))))‘2,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s_2')))))))

{-3} OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))

Hiding formulas: (-1 -2 -3 -4 -18 3),

Using lemma translate_transformer_invariant,

Using lemma expr_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -6 -7),

Using lemma subset_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1:

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')), (restrict [Address, Mem
{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))
{-6} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))‘base_addr
{-7} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-8} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-10} Mem?(data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘type_of)
{-11} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘offset
< max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
{-14} OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))
{-16} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
{-17} interpreted_data_type?(segment_reg_data_type)

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Case splitting on restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)),

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.1:`

```
{-1}  restrict[Address, Memory_Address_4G, boolean]((pm'‘ro_addr ∪ pm'‘rw_addr)) =
        (restrict[Address, Memory_Address_4G, boolean](pm'‘ro_addr) ∪ restrict [Address, Memory_Ad
{-2}  Mem?(x'‘type_of)
{-3}  0 ≤ x'‘offset
{-4}  x'‘offset < expt(2, 32)
{-5}  address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)
                                                  ((#type_of := CS_, offset := 0#))(s'_1)),
                                 restrict[Address, Memory_Address_4G, boolean]
                                     ((pm'‘ro_addr ∪ pm'‘rw_addr)))
                            (x')
-------------------------------------------------------------------------------
{1}   pm_phy‘rw_addr(x')
{2}   address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)
                                                  ((#type_of := CS_, offset := 0#))(s'_1)),
                                 (restrict[Address, Memory_Address_4G, boolean](pm'‘ro_addr) ∪ restri
                            (x')
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.1`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2:`

```
{-1}  Mem?(x'‘type_of)
{-2}  0 ≤ x'‘offset
{-3}  x'‘offset < expt(2, 32)
{-4}  address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)
                                                  ((#type_of := CS_, offset := 0#))(s'_1)),
                                 restrict[Address, Memory_Address_4G, boolean]
                                     ((pm'‘ro_addr ∪ pm'‘rw_addr)))
                            (x')
-------------------------------------------------------------------------------
{1}   restrict[Address, Memory_Address_4G, boolean]((pm'‘ro_addr ∪ pm'‘rw_addr)) =
        (restrict[Address, Memory_Address_4G, boolean](pm'‘ro_addr) ∪ restrict [Address, Memory_Ad
{2}   pm_phy‘rw_addr(x')
{3}   address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)
                                                  ((#type_of := CS_, offset := 0#))(s'_1)),
                                 (restrict[Address, Memory_Address_4G, boolean](pm'‘ro_addr) ∪ restri
                            (x')
```

Keeping (1) and hiding *,

Applying decompose-equality,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.1:`

```
{-1}  union(pm'‘ro_addr, pm'‘rw_addr)(x'')
-------------------------------------------------------------------------------
{1}   union(restrict[Address, Memory_Address_4G, boolean](pm'‘ro_addr),
              restrict[Address, Memory_Address_4G, boolean](pm'‘rw_addr))
            (x'')
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.1`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.2`:

{-1} union(restrict[Address, Memory_Address_4G, boolean](pm'ro_addr), restrict[Address, Memory_Address_4G, boolean](pm'rw_addr))(x'')

{1} union(pm'ro_addr, pm'rw_addr)(x'')

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.1.2.2`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2:`

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')), (restrict [Add

{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))

{-3} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)

{-4} is_linear_plain_memory?(pm′)

{-5} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))

{-6} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))‘base_addr

{-7} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))

{-8} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))

{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-10} Mem?(data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘type_of)

{-11} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘offset
< max_linear_offset

{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))

{-14} OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))

{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))

{-16} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))

{-17} interpreted_data_type?(segment_reg_data_type)

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2.1:

{-1}  pe_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)),
                      restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr ∪ pm'rw_addr)),
                      0)
                     (xlat_idx(0,
                               data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, segment_reg_data_type)
                                                                       (CS)
                                                                       (s'_1))))'base_addr,
                               addr'))
       ⊃
       (address_block(xlat_idx(0, data(read_data(pm_phy, pdbr_data_type)(PDBR)(state(read_data(p
{-2}  (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Add
{-3}  pm'states
          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                           (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))))
{-4}  union(pm'ro_addr, pm'rw_addr)(addr')
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'states
          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                           (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))))
{-7}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))'base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
                      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))'base_addr
{-8}  pm'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-9}  pm'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-10} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-11} Mem?(data(translate(0,
                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))'base_addr,
                          addr', ac1',
                          segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                         (s'_1))))
                      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                (s'_1))))))'2'type_of)
{-12} data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                             (s'_1))))'base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s'_1))))
                 (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                  (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s'_1))))))'2'offset
       < max_linear_offset
{-13} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-14} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-15} OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, seg-
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma xlat_pe_in_pdir_range,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2.1`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2.2:`

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')), (restrict [Add
{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))
{-6} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))‘base_addr
{-7} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-8} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-10} Mem?(data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘type_of)
{-11} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘offset
< max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
{-14} OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))
{-16} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
{-17} interpreted_data_type?(segment_reg_data_type)

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.2.2.2`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.3:
  {-1}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))‘base_addr,
                                     addr', ac1',
                                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                   (CS)
                                                                   (s'_1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))))‘2,
                      addr', ac2',
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s'_1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))‘base_addr
                                      addr', ac2',
                                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))
                                     (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy,
                                                                     segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_2))))))))
  {-2}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))‘base_addr,
                                     addr', ac1',
                                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                                   (CS)
                                                                   (s'_1))))
                                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))))‘2,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s'_1))))
                     (state(translate(0,
                                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                    (state(read_data(pm_phy, segment_reg_data_type)
                                                                    (CS)
                                                                    (s'_1))))‘base_addr
```

Hiding formulas: (-1 -2 -3 -14 3),

Using lemma translate_transformer_invariant,

Using lemma expr_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -6 -7),

Using lemma subset_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1:
{-1}  (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Add
{-2}  pm'‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                 (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))))
{-3}  union(pm'‘ro_addr, pm'‘rw_addr)(addr')
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                 (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))))
{-6}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))‘base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))‘base_addr
{-7}  pm'‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8}  pm'‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                 (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                          (s'_1))))‘base_addr,
                          addr', ac1',
                          segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                 (s'_1))))
                        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                 (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                          (s'_1))))))‘2‘type_of)
{-11} data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                     (s'_1))))‘base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                            (s'_1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                            (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                     (s'_1))))))‘2‘offset
       < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-15} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-16} OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                           (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'_1))))‘base_addr,
                    addr', ac1',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                           (s'_1))))
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                           (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                    (s'_1))))))
{-17} interpreted_data_type?(segment_reg_data_type)
```

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

Case splitting on restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)),

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.1:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})) = \\
 & \quad (\mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big](\mathrm{pm}'\text{`ro\_addr}) \cup \mathrm{restrict}\ \big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G}, \\
\{-2\} & \mathrm{Mem?}(x'\text{`type\_of}) \\
\{-3\} & 0 \le x'\text{`offset} \\
\{-4\} & x'\text{`offset} < \mathrm{expt}(2,\ 32) \\
\{-5\} & \mathrm{address\_in\_pt\_range?}(\mathrm{state}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{segment\_reg\_data\_type}) \\
 & \qquad\qquad ((\#\mathrm{type\_of} := \mathrm{CS\_},\ \mathrm{offset} := 0\#))(s_1')), \\
 & \qquad \mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big] \\
 & \qquad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr}))) \\
 & \quad (x') \\
\hline
\{1\} & \mathrm{pm\_phy}\text{`rw\_addr}(x') \\
\{2\} & \mathrm{address\_in\_pt\_range?}(\mathrm{state}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{segment\_reg\_data\_type}) \\
 & \qquad\qquad ((\#\mathrm{type\_of} := \mathrm{CS\_},\ \mathrm{offset} := 0\#))(s_1')), \\
 & \qquad (\mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big](\mathrm{pm}'\text{`ro\_addr}) \cup \mathrm{restrict}\ \big[\mathrm{Addres} \\
 & \quad (x')
\end{array}
$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.1`.

`linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.2:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{Mem?}(x'\text{`type\_of}) \\
\{-2\} & 0 \le x'\text{`offset} \\
\{-3\} & x'\text{`offset} < \mathrm{expt}(2,\ 32) \\
\{-4\} & \mathrm{address\_in\_pt\_range?}(\mathrm{state}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{segment\_reg\_data\_type}) \\
 & \qquad\qquad ((\#\mathrm{type\_of} := \mathrm{CS\_},\ \mathrm{offset} := 0\#))(s_1')), \\
 & \qquad \mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big] \\
 & \qquad\quad ((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr}))) \\
 & \quad (x') \\
\hline
\{1\} & \mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big]((\mathrm{pm}'\text{`ro\_addr} \cup \mathrm{pm}'\text{`rw\_addr})) = \\
 & \quad (\mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big](\mathrm{pm}'\text{`ro\_addr}) \cup \mathrm{restrict}\ \big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G}, \\
\{2\} & \mathrm{pm\_phy}\text{`rw\_addr}(x') \\
\{3\} & \mathrm{address\_in\_pt\_range?}(\mathrm{state}(\mathrm{read\_data}(\mathrm{pm\_phy},\ \mathrm{segment\_reg\_data\_type}) \\
 & \qquad\qquad ((\#\mathrm{type\_of} := \mathrm{CS\_},\ \mathrm{offset} := 0\#))(s_1')), \\
 & \qquad (\mathrm{restrict}\big[\mathrm{Address},\ \mathrm{Memory\_Address\_4G},\ \mathrm{boolean}\big](\mathrm{pm}'\text{`ro\_addr}) \cup \mathrm{restrict}\ \big[\mathrm{Addres} \\
 & \quad (x')
\end{array}
$$

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.1.2`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2:
{-1}  (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)), (restrict [Add
{-2}  pm'‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                 (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))))
{-3}  union(pm'‘ro_addr, pm'‘rw_addr)(addr')
{-4}  is_linear_plain_memory?(pm')
{-5}  pm'‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                 (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))))
{-6}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))‘base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))‘base_addr
{-7}  pm'‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-8}  pm'‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-9}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-10} Mem?(data(translate(0,
                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))‘base_addr,
                          addr', ac1',
                          segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))
                      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))))‘2‘type_of)
{-11} data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))‘base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))))‘2‘offset
       < max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-14} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-15} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
             (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-16} OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))‘base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'_1))))))
{-17} interpreted_data_type?(segment_reg_data_type)
```

Using lemma pe_in_pt_address_in_pt,

we get 2 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.1:

{-1}  pe_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)),
                      restrict[Address, Memory_Address_4G, boolean]
                          ((pm'ro_addr ∪ pm'rw_addr)),
                      0)
                     (xlat_idx(0,
                               data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, segment_reg_data_type)
                                                                       (CS)
                                                                       (s'1))))'base_addr,
                               addr'))
       ⊃
       (address_block(xlat_idx(0, data(read_data(pm_phy, pdbr_data_type)(PDBR)(state(read_data(p
{-2}  (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)), (restrict [Add
{-3}  pm'states
          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                           (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)))))
{-4}  union(pm'ro_addr, pm'rw_addr)(addr')
{-5}  is_linear_plain_memory?(pm')
{-6}  pm'states
          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                           (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2)))))
{-7}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1))))'base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
                      (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))))'base_addr
{-8}  pm'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2)))
{-9}  pm'states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)))
{-10} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))
{-11} Mem?(data(translate(0,
                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                  (s'1))))'base_addr,
                          addr', ac1',
                          segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                         (s'1))))
                      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                (s'1))))))'2'type_of)
{-12} data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                             (s'1))))'base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s'1))))
                 (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                  (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                           (s'1))))))'2'offset
       < max_linear_offset
{-13} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))
{-14} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))))
{-15} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'1))
{-16} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma xlat_pe_in_pdir_range,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.1`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.2:
```

{-1} (address_in_pt_range?(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')), (restrict [Add
{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))
{-6} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))‘base_addr
{-7} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-8} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-9} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-10} Mem?(data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘type_of)
{-11} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘offset
< max_linear_offset
{-12} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-13} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
{-14} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))
{-15} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
{-16} OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
{-17} interpreted_data_type?(segment_reg_data_type)

Using lemma xlat_idx_memory_address,

Using lemma xlat_idx_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.1.3.2.2`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.2:
```

{-1} OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))))‘2,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))‘base_addr
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s_1'))))))))

{-2} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))

=

data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)

Revealing hidden formulas,

Hiding formulas: (-2 -5 -6 3),

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_resolve_same_result.1.1.1.1.1.1.2.2.1:`

{-1} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))‘1

{-2} OK?(OK(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_2')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s_2'))))),
xlat_ofs(0,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data
(pm_phy, segment_reg_data_type)
(CS)
(s_2')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s_2')))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data
(pm_phy, segment_reg_data_type)
(CS)
(s_2'))))))‘2,
addr′)))

{-3} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))

=

Replacing using formula -3,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.2.1`.

```
linear_resolve_same_result.1.1.1.1.1.1.2.2.2:
  {-1}  OK?(translate(1,
                      data(translate(0,
                                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy, segment_reg_data_type)
                                                 (CS)
                                                 (s'_2))))‘base_addr,
                                     addr', ac2',
                                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                          (CS)
                                                          (s'_2))))
                                (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, segment_reg_data_type)
                                              (CS)
                                              (s'_2))))))‘2,
                      addr', ac2',
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                           (s'_2))))
                (state(translate(0,
                                 data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                      (state(read_data(pm_phy, segment_reg_data_type)
                                             (CS)
                                             (s'_2))))‘base_addr
                                 addr', ac2',
                                 segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                      (CS)
                                                      (s'_2))))
                            (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy,
                                                    segment_reg_data_type)
                                          (CS)
                                          (s'_2))))))))
  {-2}  OK?(OK(state(translate(1,
                               data(translate(0,
                                              data(read_data(pm_phy, pdbr_data_type)(P
                                                   (state
                                                     (read_data
                                                       (pm_phy, segment_reg_data_type)
                                                       (CS)
                                                       (s'_2))))‘base_addr,
                                              addr', ac2',
                                              segment_to_priv(data
                                                              (read_data
                                                                (pm_phy, segment_reg_data_type)
                                                                (CS)
                                                                (s'_2))))
                                         (state(read_data(pm_phy, pdbr_data_type)
                                                         (PDBR)
                                                         (state
                                                           (read_data
                                                             (pm_phy, segment_reg_data_type)
                                                             (CS)
                                                             (s'_2))))))‘2,
                               addr', ac2',
                               segment_to_priv(data(read_data(pm_phy, seg-
```

Replacing using formula -3,

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.2.2`.

linear_resolve_same_result.1.1.1.1.1.1.2.3:

{-1} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
=
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))

{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))

{-3} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))

{-4} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))‘base_addr

{-5} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))

{-6} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))

{-7} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-8} data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2‘type_of
= Mem_
∧
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,

Hiding formulas: 3,

Revealing hidden formulas,

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.1.2.3`.

```
linear_resolve_same_result.1.1.1.1.1.2:

{-1}  pm′‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                         (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2)))))
{-2}  pm′‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                         (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)))))
{-3}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1))))‘base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2))))‘base_addr
{-4}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2)))
{-5}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)))
{-6}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s′2))
{-7}  aligned?(bits_per_level + pe_size)
          (offset
              (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                             (s′1))))‘base_addr))
       ∧
       OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                   (s′1))))‘base_addr,
                     addr′, ac1′,
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                 (s′1))))
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                   (s′1))))))
       ⊃
       data(translate(0,
                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                    (s′1))))‘base_addr,
                      addr′, ac1′,
                      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                  (s′1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                    (s′1))))))‘2
         < max_linear
{-8}  aligned?(bits_per_level + pe_size)
          (offset
              (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                             (s′2))))‘base_addr))
       ∧
1972   OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                   (s′2))))‘base_addr,
                     addr′, ac2′,
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                  (s′2))))
```

Hiding formulas: 2,

Using lemma translate_same_result,

we get 2 subgoals:

```
linear_resolve_same_result.1.1.1.1.1.2.1:

{-1}  union(pm′‘ro_addr, pm′‘rw_addr)(addr′) ∧
       is_linear_plain_memory?(pm′) ∧
        pe_in_pt_range?(state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′1)))),
                        restrict[Address, Memory_Address_4G, boolean]
                            ((pm′‘ro_addr ∪ pm′‘rw_addr)),
                        0)
                       (xlat_idx(0,
                                 data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                 (state(read_data(pm_phy, segment_reg_data_type)
                                                                 (CS)
                                                                 (s′1)))‘base_addr,
                                 addr′))
          ∧
          OK?(translate(0,
                        data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′1)))‘base_addr,
                        addr′, ac1′,
                        segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                      (s′1))))
                       (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′1))))))
           ∧
           OK?(translate(0,
                         data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                         (s′1)))‘base_addr,
                         addr′, ac2′,
                         segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                       (s′1))))
                        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                         (s′2))))))
        ⊃
        data(translate(0,
                       data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                       (s′1)))‘base_addr,
                       addr′, ac1′,
                       segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                     (s′1))))
                      (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                       (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                       (s′1))))))
          =
          data(translate(0,
                         data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)(CS)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -4 -5 -6 -7 -8 -9 -10 2),

Using lemma xlat_pe_in_pdir_range,

Expanding the definition of pe_in_pt_range?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.2.1`.

```
linear_resolve_same_result.1.1.1.1.1.2.2:

{-1}  pm′‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                         (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2)))))
{-2}  pm′‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                         (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)))))
{-3}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1))))‘base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
                (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2))))‘base_addr
{-4}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2)))
{-5}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)))
{-6}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s′2))
{-7}  aligned?(bits_per_level + pe_size)
          (offset
              (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                             (s′1))))‘base_addr))
       ∧
       OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                   (s′1))))‘base_addr,
                     addr′, ac1′,
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                 (s′1))))
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                   (s′1))))))
       ⊃
       data(translate(0,
                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                    (s′1))))‘base_addr,
                      addr′, ac1′,
                      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                  (s′1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                    (s′1))))))‘2
         < max_linear
{-8}  aligned?(bits_per_level + pe_size)
          (offset
              (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                              (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                             (s′2))))‘base_addr))
       ∧
1976   OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                   (s′2))))‘base_addr,
                     addr′, ac2′,
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                 (s′2))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.2.2`.

```
linear_resolve_same_result.1.1.1.1.1.3:
```

{-1} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))
{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))
{-3} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))‘base_addr
{-4} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-5} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-7} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear
{-8} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr))
∧
1978 OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.3`.

```
linear_resolve_same_result.1.1.1.1.1.4:
```

{-1} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))))
{-2} pm′‘states
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))))
{-3} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))‘base_addr
{-4} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-5} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-6} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-7} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear
{-8} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.1.4`.

```
linear_resolve_same_result.1.1.1.1.2:

{-1}  pm'‘states
        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                         (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)))))
{-2}  data(read_data(pm_phy, pdbr_data_type)(PDBR)
                   (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1))))‘base_addr
       =
       data(read_data(pm_phy, pdbr_data_type)(PDBR)
                   (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))))‘base_addr
{-3}  pm'‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2)))
{-4}  pm'‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)))
{-5}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))
{-6}  aligned?(bits_per_level + pe_size)
            (offset
                (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                 (s'1))))‘base_addr))
       ∧
       OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                     (s'1))))‘base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                     (s'1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                     (s'1))))))
       ⊃
       data(translate(0,
                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                      (s'1))))‘base_addr,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                      (s'1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                      (s'1))))))‘2
         < max_linear
{-7}  aligned?(bits_per_level + pe_size)
            (offset
                (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                 (s'2))))‘base_addr))
       ∧
       OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                     (s'2))))‘base_addr,
                     addr', ac2',
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                     (s'2))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, seg-
```

Hiding formulas: (-1 -2 -6 -7 -10 -13 2),

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.2`.

```
linear_resolve_same_result.1.1.1.1.3:
```

{-1} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))‘base_addr
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))‘base_addr

{-2} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))

{-3} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))

{-4} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

{-5} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear

{-6} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2')))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
⊃

Hiding formulas: (-1 -5 -6 -9 -12 2),

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.3`.

```
linear_resolve_same_result.1.1.1.1.4:

{-1}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′2)))
{-2}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)))
{-3}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s′2))
{-4}  aligned?(bits_per_level + pe_size)
            (offset
                (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′1))))‘base_addr))
       ∧
       OK?(translate(0,
                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                  (s′1))))‘base_addr,
                          addr′, ac1′,
                          segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                (s′1))))
                        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                  (s′1))))))
       ⊃
       data(translate(0,
                           data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                  (s′1))))‘base_addr,
                           addr′, ac1′,
                           segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                (s′1))))
                         (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                  (s′1))))))‘2
        < max_linear
{-5}  aligned?(bits_per_level + pe_size)
            (offset
                (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′2))))‘base_addr))
       ∧
       OK?(translate(0,
                          data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                  (s′2))))‘base_addr,
                          addr′, ac2′,
                          segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                (s′2))))
                        (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
1986  ment_reg_data_type)(CS)
                                                                  (s′2))))))
       ⊃
       data(translate(0,
                           data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                              (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                  (s′2))))‘base_addr,
```

Hiding formulas: (-4 -5 -8 -11 2),

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -18 with the terms: (state(read_data(pm_phy, segment_reg_data_type)(CS)(s1!1)) state(read_data(pm_phy, segment_reg_data_type)(CS)(s2!1))),

we get 3 subgoals:

`linear_resolve_same_result.1.1.1.1.4.1:`

{-1} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)))
{-2} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-4} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-5} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{-8} interpreted_data_type?(segment_reg_data_type)
{-9} interpreted_data_type?(pdbr_data_type)
{-10} Mem?(addr′‘type_of)
{-11} $0 \leq$ addr′‘offset
{-12} addr′‘offset $<$ max_linear_offset
{-13} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-14} pm′‘mem = linear_pm
{-15} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-16} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-17} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-18} data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
=
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
{-19} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-20} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-21} ∀ (s_1, s_2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-22} plain_memory?(pm_phy)
{-23} pm′‘states = pm_phy‘states
{-24} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-25} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-26} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ $0 \leq$ offset(a) ∧ offset(a) $<$ max_linear_offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.1.4.1`.

linear_resolve_same_result.1.1.1.1.4.2:

{-1} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-2} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-4} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-5} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
{-8} interpreted_data_type?(segment_reg_data_type)
{-9} interpreted_data_type?(pdbr_data_type)
{-10} Mem?(addr′‘type_of)
{-11} $0 \leq$ addr′‘offset
{-12} addr′‘offset $<$ max_linear_offset
{-13} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-14} pm′‘mem = linear_pm
{-15} $\forall$ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-16} $\forall$ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-17} $\forall$ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-18} $\forall$ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr $\cup$ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr $\cup$ pm′‘rw_addr)))
$\wedge$
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr $\cup$ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr $\cup$ pm′‘rw_addr)))
{-19} $\forall$ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr $\cup$ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-20} $\forall$ (s_1, s_2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr $\cup$ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-21} plain_memory?(pm_phy)
{-22} pm′‘states = pm_phy‘states
{-23} (pm′‘other_actions $\subseteq$ pm_phy‘other_actions)
{-24} (linear_resolve_register_transformers $\subseteq$ pm_phy‘other_actions)
{-25} $\forall$ (a: ((pm′‘ro_addr $\cup$ pm′‘rw_addr))):
Mem?(type_of(a)) $\wedge$ $0 \leq$ offset(a) $\wedge$ offset(a) $<$ max_linear_offset
{-26} $\forall$ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-27} pm′‘states(s_1')
{-28} pm′‘states(s_2')

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.4.2`.

`linear_resolve_same_result.1.1.1.1.4.3:`

{-1} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')))
{-2} pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')))
{-3} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-4} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-5} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
{-8} interpreted_data_type?(segment_reg_data_type)
{-9} interpreted_data_type?(pdbr_data_type)
{-10} Mem?(addr′‘type_of)
{-11} $0 \leq$ addr′‘offset
{-12} addr′‘offset $<$ max_linear_offset
{-13} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-14} pm′‘mem = linear_pm
{-15} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-16} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1)) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2))
{-17} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-18} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-19} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-20} ∀ (s_1, s_2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-21} plain_memory?(pm_phy)
{-22} pm′‘states = pm_phy‘states
{-23} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-24} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-25} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ $0 \leq$ offset(a) ∧ offset(a) $<$ max_linear_offset
{-26} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-27} pm′‘states(s_1')
{-28} pm′‘states(s_2')

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.1.4.3`.

```
linear_resolve_same_result.1.1.1.2:

{-1}  pm′‘states(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)))
{-2}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)) =
      data(read_data(pm_phy, segment_reg_data_type)(CS)(s′2))
{-3}  aligned?(bits_per_level + pe_size)
            (offset
                (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′1))))‘base_addr))
      ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                            (s′1))))‘base_addr,
                    addr′, ac1′,
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                        (s′1))))
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                            (s′1))))))
      ⊃
      data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                            (s′1))))‘base_addr,
                     addr′, ac1′,
                     segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                        (s′1))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                            (s′1))))))‘2
        < max_linear
{-4}  aligned?(bits_per_level + pe_size)
            (offset
                (data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                        (s′2))))‘base_addr))
      ∧
      OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                            (s′2))))‘base_addr,
                    addr′, ac2′,
                    segment_to_priv(data(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                        (s′2))))
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
1994                                                    (s′2))))))
      ⊃
      data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, seg-
ment_reg_data_type)(CS)
                                                            (s′2))))‘base_addr,
                     addr′, ac2′,
```

Hiding formulas: (-1 -2 -3 -4 -6 -7 -9 -10 2),

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.2`.

linear_resolve_same_result.1.1.1.3:

{-1} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-2} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear
{-3} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, seg-

Hiding formulas: (-1 -2 -3 -5 -6 -8 -9 2),

Using lemma pm_unchanged_singleton_linear_resolve_reg_transformers,

Expanding the definition of linear_resolve_register_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma unchanged_memory_invariant_invariant,

Using lemma expr_transformer_invariant_next_ok,

Using lemma pm_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.1.3`.

```
linear_resolve_same_result.1.1.1.4:
```

{-1} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
⊃
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))‘2
< max_linear

{-2} aligned?(bits_per_level + pe_size)
(offset
(data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
⊃
data(translate(0,
1998 data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))‘base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))

Hiding formulas: (-1 -2 -4 -5 -7 -8 2),

Expanding the definition of is_linear_plain_memory?,

Applying disjunctive simplification to flatten sequent,

Instantiating the top quantifier in -11 with the terms: (s1!1 s2!1),

we get 3 subgoals:

`linear_resolve_same_result.1.1.1.4.1:`

{-1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-2} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))
{-3} interpreted_data_type?(segment_reg_data_type)
{-4} interpreted_data_type?(pdbr_data_type)
{-5} Mem?(addr′‘type_of)
{-6} 0 ≤ addr′‘offset
{-7} addr′‘offset < max_linear_offset
{-8} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-9} pm′‘mem = linear_pm
{-10} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))
{-12} ∀ (s: (pm′‘states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-13} ∀ (s_1, s_2: (pm′‘states)):
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-14} ∀ (s_1, s_2: (pm′‘states)):
pe_in_pdir_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_pdir_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
∧
pe_in_ptab_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
=
pe_in_ptab_range?(s_2,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-15} ∀ (s: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-16} ∀ (s_1, s_2: (pm′‘states), lvl: Level,
a:
(pe_in_pt_range?(s_1,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)),
lvl))):
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-17} plain_memory?(pm_phy)
{-18} pm′‘states = pm_phy‘states
{-19} (pm′‘other_actions ⊆ pm_phy‘other_actions)
{-20} (linear_resolve_register_transformers ⊆ pm_phy‘other_actions)
{-21} ∀ (a: ((pm′‘ro_addr ∪ pm′‘rw_addr))):
Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-22} ∀ (s: (pm′‘states)):
linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
{-23} pm′‘states(s_1')
{-24} pm′‘states(s_2')

{1} data(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) =
data(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.4.1`.

`linear_resolve_same_result.1.1.1.4.2:`

```
{-1}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-2}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-3}  interpreted_data_type?(segment_reg_data_type)
{-4}  interpreted_data_type?(pdbr_data_type)
{-5}  Mem?(addr'`type_of)
{-6}  0 ≤ addr'`offset
{-7}  addr'`offset < max_linear_offset
{-8}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-9}  pm'`mem = linear_pm
{-10} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s_1, s_2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-13} ∀ (s_1, s_2: (pm'`states)):
        pe_in_pdir_range?(s_1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'`ro_addr ∪ pm'`rw_addr)))
         =
         pe_in_pdir_range?(s_2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'`ro_addr ∪ pm'`rw_addr)))
         ∧
         pe_in_ptab_range?(s_1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'`ro_addr ∪ pm'`rw_addr)))
          =
          pe_in_ptab_range?(s_2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm'`ro_addr ∪ pm'`rw_addr)))
{-14} ∀ (s: (pm'`states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'`ro_addr ∪ pm'`rw_addr)),
                              lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-15} ∀ (s_1, s_2: (pm'`states), lvl: Level,
           a:
             (pe_in_pt_range?(s_1,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'`ro_addr ∪ pm'`rw_addr)),
                              lvl))):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-16} plain_memory?(pm_phy)
{-17} pm'`states = pm_phy`states
{-18} (pm'`other_actions ⊆ pm_phy`other_actions)
{-19} (linear_resolve_register_transformers ⊆ pm_phy`other_actions)
{-20} ∀ (a: ((pm'`ro_addr ∪ pm'`rw_addr))):
        Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-21} ∀ (s: (pm'`states)):
        linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
                           restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
{-22} pm'`states(s'_1)
{-23} pm'`states(s'_2)
------------------------------------------------------------------------
 {1}  pm'`states(s'_2)
 {2}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
```

which is trivially true.

This completes the proof of `linear_resolve_same_result.1.1.1.4.2`.

`linear_resolve_same_result.1.1.1.4.3:`

```
{-1}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
{-2}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))
{-3}  interpreted_data_type?(segment_reg_data_type)
{-4}  interpreted_data_type?(pdbr_data_type)
{-5}  Mem?(addr'`type_of)
{-6}  0 ≤ addr'`offset
{-7}  addr'`offset < max_linear_offset
{-8}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-9}  pm'`mem = linear_pm
{-10} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s))
{-11} ∀ (s: (pm'`states)): OK?(read_data(pm_phy, pdbr_data_type)(PDBR)(s))
{-12} ∀ (s_1, s_2: (pm'`states)):
        data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_1)) =
         data(read_data(pm_phy, pdbr_data_type)(PDBR)(s_2))
{-13} ∀ (s_1, s_2: (pm'`states)):
        pe_in_pdir_range?(s_1,
                          restrict[Address, Memory_Address_4G, boolean]
                              ((pm'`ro_addr ∪ pm'`rw_addr)))
         =
         pe_in_pdir_range?(s_2,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'`ro_addr ∪ pm'`rw_addr)))
         ∧
         pe_in_ptab_range?(s_1,
                           restrict[Address, Memory_Address_4G, boolean]
                               ((pm'`ro_addr ∪ pm'`rw_addr)))
          =
          pe_in_ptab_range?(s_2,
                            restrict[Address, Memory_Address_4G, boolean]
                                ((pm'`ro_addr ∪ pm'`rw_addr)))
{-14} ∀ (s: (pm'`states), lvl: Level,
           a:
             (pe_in_pt_range?(s,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'`ro_addr ∪ pm'`rw_addr)),
                              lvl))):
        OK?(read_data(pm_phy, paging_data_type(lvl))(a)(s))
{-15} ∀ (s_1, s_2: (pm'`states), lvl: Level,
           a:
             (pe_in_pt_range?(s_1,
                              restrict[Address, Memory_Address_4G, boolean]
                                  ((pm'`ro_addr ∪ pm'`rw_addr)),
                              lvl))):
        set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_1)), Write) =
         set_reference(data(read_data(pm_phy, paging_data_type(lvl))(a)(s_2)), Write)
{-16} plain_memory?(pm_phy)
{-17} pm'`states = pm_phy`states
{-18} (pm'`other_actions ⊆ pm_phy`other_actions)
{-19} (linear_resolve_register_transformers ⊆ pm_phy`other_actions)
{-20} ∀ (a: ((pm'`ro_addr ∪ pm'`rw_addr))):
        Mem?(type_of(a)) ∧ 0 ≤ offset(a) ∧ offset(a) < max_linear_offset
{-21} ∀ (s: (pm'`states)):
        linear_blessed?(s, restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
                           restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
{-22} pm'`states(s'_1)
{-23} pm'`states(s'_2)
------------------------------------------------------------------------
 {1}  pm'`states(s'_1)
 {2}  data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) =
       data(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))
```

which is trivially true.
This completes the proof of `linear_resolve_same_result.1.1.1.4.3`.
`linear_resolve_same_result.1.1.2`:

{-1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) ∧
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))`base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))
{-2} interpreted_data_type?(segment_reg_data_type)
{-3} interpreted_data_type?(pdbr_data_type)
{-4} Mem?(addr′`type_of)
{-5} 0 ≤ addr′`offset
{-6} addr′`offset < max_linear_offset
{-7} union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-8} is_linear_plain_memory?(pm′)
{-9} pm′`states(s_1')
{-10} pm′`states(s_2')
{-11} OK?(linear_resolve(addr′, ac1′)(s_1'))
{-12} OK?(linear_resolve(addr′, ac2′)(s_2'))

{1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')) ∧
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_2'))))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))`base_addr,
addr′, ac2′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_2'))))))
{2} data(linear_resolve(addr′, ac1′)(s_1')) = data(linear_resolve(addr′, ac2′)(s_2'))

Hiding formulas: (-1 -8 2),
Expanding the definition of linear_resolve,
Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.1.1.2`.
`linear_resolve_same_result.1.1.3`:

```
{-1}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) ∧
       OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
        ∧
        OK?(translate(0,
                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                          (s'_1))))`base_addr,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                          (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                                          (s'_1))))))
{-2}  interpreted_data_type?(segment_reg_data_type)
{-3}  interpreted_data_type?(pdbr_data_type)
{-4}  Mem?(addr'`type_of)
{-5}  0 ≤ addr'`offset
{-6}  addr'`offset < max_linear_offset
{-7}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-8}  is_linear_plain_memory?(pm')
{-9}  pm'`states(s'_1)
{-10} pm'`states(s'_2)
{-11} OK?(linear_resolve(addr', ac1')(s'_1))
{-12} OK?(linear_resolve(addr', ac2')(s'_2))
-------------------------------------------------------------------------------
 {1}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2)) ∧
       OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_2))))
       ⊃
       OK?[Physical_memory, Pdbr_type]
           (read_data[Physical_memory, Pdbr_type]
                     (pm_phy, pdbr_data_type)(PDBR)
                     (state[Physical_memory, Segment_Reg_type]
                           (read_data[Physical_memory, Segment_Reg_type]
                                     (pm_phy, segment_reg_data_type)(CS)(s'_2))))
        ∨
        Exception?[Physical_memory, Pdbr_type]
            (read_data[Physical_memory, Pdbr_type]
                      (pm_phy, pdbr_data_type)(PDBR)
                      (state[Physical_memory, Segment_Reg_type]
                            (read_data[Physical_memory, Segment_Reg_type]
                                      (pm_phy, segment_reg_data_type)(CS)(s'_2))))
 {2}  data(linear_resolve(addr', ac1')(s'_1)) = data(linear_resolve(addr', ac2')(s'_2))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.1.1.3`.

`linear_resolve_same_result.1.1.4:`

```
{-1}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'1)) ∧
       OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'1))))
        ∧
       OK?(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                   (s'1))))`base_addr,
                     addr', ac1',
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                   (s'1))))
                                (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                               (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                               (s'1))))))
{-2}  interpreted_data_type?(segment_reg_data_type)
{-3}  interpreted_data_type?(pdbr_data_type)
{-4}  Mem?(addr'`type_of)
{-5}  0 ≤ addr'`offset
{-6}  addr'`offset < max_linear_offset
{-7}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-8}  is_linear_plain_memory?(pm')
{-9}  pm'`states(s'1)
{-10} pm'`states(s'2)
{-11} OK?(linear_resolve(addr', ac1')(s'1))
{-12} OK?(linear_resolve(addr', ac2')(s'2))
-------------------------------------------------------------------------
 {1}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'2)) ∧
       OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'2))))
       ⊃
       OK?[Physical_memory, Segment_Reg_type]
          (read_data[Physical_memory, Segment_Reg_type]
                    (pm_phy, segment_reg_data_type)(CS)(s'2))
        ∨
        Exception?[Physical_memory, Segment_Reg_type]
           (read_data[Physical_memory, Segment_Reg_type]
                     (pm_phy, segment_reg_data_type)(CS)(s'2))
 {2}  data(linear_resolve(addr', ac1')(s'1)) = data(linear_resolve(addr', ac2')(s'2))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.4`.

`linear_resolve_same_result.1.1.5:`

{-1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_1')) ∧
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s_1'))))
∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1')))‘base_addr,
addr′, ac1′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s_1'))))))

{-2} interpreted_data_type?(segment_reg_data_type)

{-3} interpreted_data_type?(pdbr_data_type)

{-4} Mem?(addr′‘type_of)

{-5} $0 \leq$ addr′‘offset

{-6} addr′‘offset $<$ max_linear_offset

{-7} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)

{-8} is_linear_plain_memory?(pm′)

{-9} pm′‘states(s_1')

{-10} pm′‘states(s_2')

{-11} OK?(linear_resolve(addr′, ac1′)(s_1'))

{-12} OK?(linear_resolve(addr′, ac2′)(s_2'))

{1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s_2')) ⊃
OK?[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s_2'))
∨
Exception?[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s_2'))

{2} data(linear_resolve(addr′, ac1′)(s_1')) = data(linear_resolve(addr′, ac2′)(s_2'))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.1.5`.

`linear_resolve_same_result.1.2:`

```
{-1}  interpreted_data_type?(segment_reg_data_type)
{-2}  interpreted_data_type?(pdbr_data_type)
{-3}  Mem?(addr'`type_of)
{-4}  0 ≤ addr'`offset
{-5}  addr'`offset < max_linear_offset
{-6}  union(pm'`ro_addr, pm'`rw_addr)(addr')
{-7}  is_linear_plain_memory?(pm')
{-8}  pm'`states(s'_1)
{-9}  pm'`states(s'_2)
{-10} OK?(linear_resolve(addr', ac1')(s'_1))
{-11} OK?(linear_resolve(addr', ac2')(s'_2))
-------------------------------------------------------------------------------
{1}   OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) ∧
       OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                     (state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
        ∧
        OK?(translate(0,
                      data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                     (s'_1))))`base_addr,
                      addr', ac1',
                      segment_to_priv(data(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                   (s'_1))))
                    (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                     (state(read_data(pm_phy, seg-
      ment_reg_data_type)(CS)
                                                                     (s'_1))))))
{2}   data(linear_resolve(addr', ac1')(s'_1)) = data(linear_resolve(addr', ac2')(s'_2))
```

Hiding formulas: (-8 2),

Expanding the definition of linear_resolve,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.2`.

`linear_resolve_same_result.1.3:`

{-1} interpreted_data_type?(segment_reg_data_type)
{-2} interpreted_data_type?(pdbr_data_type)
{-3} Mem?(addr′‘type_of)
{-4} $0 \leq$ addr′‘offset
{-5} addr′‘offset $<$ max_linear_offset
{-6} union(pm′‘ro_addr, pm′‘rw_addr)(addr′)
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s'_1)
{-9} pm′‘states(s'_2)
{-10} OK?(linear_resolve(addr′, ac1′)(s'_1))
{-11} OK?(linear_resolve(addr′, ac2′)(s'_2))

{1} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1)) ∧
OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s'_1))))
⊃
OK?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s'_1))))
∨
Exception?[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s'_1))))
{2} data(linear_resolve(addr′, ac1′)(s'_1)) = data(linear_resolve(addr′, ac2′)(s'_2))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_result.1.3`.

`linear_resolve_same_result.1.4`:

```
  {-1}  interpreted_data_type?(segment_reg_data_type)
  {-2}  interpreted_data_type?(pdbr_data_type)
  {-3}  Mem?(addr′`type_of)
  {-4}  0 ≤ addr′`offset
  {-5}  addr′`offset < max_linear_offset
  {-6}  union(pm′`ro_addr, pm′`rw_addr)(addr′)
  {-7}  is_linear_plain_memory?(pm′)
  {-8}  pm′`states(s′1)
  {-9}  pm′`states(s′2)
 {-10}  OK?(linear_resolve(addr′, ac1′)(s′1))
 {-11}  OK?(linear_resolve(addr′, ac2′)(s′2))
-------------------------------------------------------------------
  {1}   OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)) ∧
         OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                       (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′1))))
         ⊃
         OK?[Physical_memory, Segment_Reg_type]
             (read_data[Physical_memory, Segment_Reg_type]
                       (pm_phy, segment_reg_data_type)(CS)(s′1))
          ∨
          Exception?[Physical_memory, Segment_Reg_type]
                (read_data[Physical_memory, Segment_Reg_type]
                          (pm_phy, segment_reg_data_type)(CS)(s′1))
  {2}   data(linear_resolve(addr′, ac1′)(s′1)) = data(linear_resolve(addr′, ac2′)(s′2))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.1.4`.

`linear_resolve_same_result.1.5`:

```
  {-1}  interpreted_data_type?(segment_reg_data_type)
  {-2}  interpreted_data_type?(pdbr_data_type)
  {-3}  Mem?(addr′`type_of)
  {-4}  0 ≤ addr′`offset
  {-5}  addr′`offset < max_linear_offset
  {-6}  union(pm′`ro_addr, pm′`rw_addr)(addr′)
  {-7}  is_linear_plain_memory?(pm′)
  {-8}  pm′`states(s′1)
  {-9}  pm′`states(s′2)
 {-10}  OK?(linear_resolve(addr′, ac1′)(s′1))
 {-11}  OK?(linear_resolve(addr′, ac2′)(s′2))
-------------------------------------------------------------------
  {1}   OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s′1)) ⊃
         OK?[Physical_memory, Segment_Reg_type]
             (read_data[Physical_memory, Segment_Reg_type]
                       (pm_phy, segment_reg_data_type)(CS)(s′1))
          ∨
          Exception?[Physical_memory, Segment_Reg_type]
                (read_data[Physical_memory, Segment_Reg_type]
                          (pm_phy, segment_reg_data_type)(CS)(s′1))
  {2}   data(linear_resolve(addr′, ac1′)(s′1)) = data(linear_resolve(addr′, ac2′)(s′2))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.1.5`.

`linear_resolve_same_result.2:`

{-1}	interpreted_data_type?(pdbr_data_type)
{-2}	Mem?(addr′`type_of)
{-3}	$0 \leq$ addr′`offset
{-4}	addr′`offset $<$ max_linear_offset
{-5}	union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-6}	is_linear_plain_memory?(pm′)
{-7}	pm′`states(s_1')
{-8}	pm′`states(s_2')
{-9}	OK?(linear_resolve(addr′, ac1′)(s_1'))
{-10}	OK?(linear_resolve(addr′, ac2′)(s_2'))
{1}	interpreted_data_type?(segment_reg_data_type)
{2}	data(linear_resolve(addr′, ac1′)(s_1')) $=$ data(linear_resolve(addr′, ac2′)(s_2'))

Adding type constraints for segment_reg_data_type,
Expanding the definition of pod_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.2`.
`linear_resolve_same_result.3:`

{-1}	Mem?(addr′`type_of)
{-2}	$0 \leq$ addr′`offset
{-3}	addr′`offset $<$ max_linear_offset
{-4}	union(pm′`ro_addr, pm′`rw_addr)(addr′)
{-5}	is_linear_plain_memory?(pm′)
{-6}	pm′`states(s_1')
{-7}	pm′`states(s_2')
{-8}	OK?(linear_resolve(addr′, ac1′)(s_1'))
{-9}	OK?(linear_resolve(addr′, ac2′)(s_2'))
{1}	interpreted_data_type?(pdbr_data_type)
{2}	data(linear_resolve(addr′, ac1′)(s_1')) $=$ data(linear_resolve(addr′, ac2′)(s_2'))

Adding type constraints for pdbr_data_type,
Expanding the definition of pod_data_type?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_result.3`.
Q.E.D.

C.117.56 Linear_Memory_Properties.delta_memory_address

Terse proof for `delta_memory_address`.
`delta_memory_address:`

{1}	$\forall$ (addr: Memory_Address_4G, δ: nat): rem(expt(2, min_page))(offset(addr)) $+ \delta <$ expt(2, min_page) $\supset$ offset(addr $+ \delta$) $<$ max_linear_offset

Repeatedly Skolemizing and flattening,
Expanding the definition of $+$,
Using lemma aligned_add_below,
we get 2 subgoals:

`delta_memory_address.1`:

{-1} $\text{min\_page} \leq \text{bus\_width} \wedge$
$\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) < \text{expt}(2,\ \text{bus\_width})$
$\wedge$
$\text{aligned?}(\text{min\_page})(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')))$
$\wedge\ \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$
$\supset$
$\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}'$
$< \text{expt}(2,\ \text{bus\_width})$
{-2} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-3} $0 \leq \text{addr}'\text{`offset}$
{-4} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-5} $\text{delta}' \geq 0$
{-6} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $\text{addr}'\text{`offset} + \text{delta}' < \text{max\_linear\_offset}$

Installing automatic rewrites from: (min_page! bus_width! max_linear_offset!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma rem_def2,

Expanding the definition of aligned?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `delta_memory_address.1`.

`delta_memory_address.2`:

{-1} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-2} $0 \leq \text{addr}'\text{`offset}$
{-3} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-4} $\text{delta}' \geq 0$
{-5} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) \geq 0$
{2} $\text{addr}'\text{`offset} + \text{delta}' < \text{max\_linear\_offset}$

Using lemma rem_floor,

Case splitting on expt(2, min_page) * floor(offset(addr!1) / expt(2, min_page)) >= 0,

we get 2 subgoals:

`delta_memory_address.2.1`:

{-1} $\text{expt}(2,\ \text{min\_page}) \times \text{floor}(\text{offset}(\text{addr}')/\text{expt}(2,\ \text{min\_page})) \geq 0$
{-2} $\text{offset}(\text{addr}') =$
$\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{expt}(2,\ \text{min\_page}) \times \text{floor}(\text{offset}(\text{addr}')/\text{expt}(2,\ \text{min\_page}))$
{-3} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-4} $0 \leq \text{addr}'\text{`offset}$
{-5} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-6} $\text{delta}' \geq 0$
{-7} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) \geq 0$
{2} $\text{addr}'\text{`offset} + \text{delta}' < \text{max\_linear\_offset}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `delta_memory_address.2.1`.

`delta_memory_address.2.2:`

{-1}	offset(addr′) = rem(expt(2, min_page))(offset(addr′)) + expt(2, min_page) × floor(offset(addr′)/expt(2, min_pag
{-2}	Mem?(addr′`type_of)
{-3}	0 ≤ addr′`offset
{-4}	addr′`offset < max_linear_offset
{-5}	delta′ ≥ 0
{-6}	rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{1}	expt(2, min_page) × floor(offset(addr′)/expt(2, min_page)) ≥ 0
{2}	offset(addr′) − rem(expt(2, min_page))(offset(addr′)) ≥ 0
{3}	addr′`offset + delta′ < max_linear_offset

Using lemma nonneg_floor_is_nat,
we get 2 subgoals:
`delta_memory_address.2.2.1:`

{-1}	floor(offset(addr′)/expt(2, min_page)) ≥ 0
{-2}	offset(addr′) = rem(expt(2, min_page))(offset(addr′)) + expt(2, min_page) × floor(offset(addr′)/expt(2, min_pag
{-3}	Mem?(addr′`type_of)
{-4}	0 ≤ addr′`offset
{-5}	addr′`offset < max_linear_offset
{-6}	delta′ ≥ 0
{-7}	rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{1}	expt(2, min_page) × floor(offset(addr′)/expt(2, min_page)) ≥ 0
{2}	offset(addr′) − rem(expt(2, min_page))(offset(addr′)) ≥ 0
{3}	addr′`offset + delta′ < max_linear_offset

Rewriting using nnreal_times_nnreal_is_nnreal, matching in *,
This completes the proof of `delta_memory_address.2.2.1`.
`delta_memory_address.2.2.2:`

{-1}	offset(addr′) = rem(expt(2, min_page))(offset(addr′)) + expt(2, min_page) × floor(offset(addr′)/expt(2, min_pag
{-2}	Mem?(addr′`type_of)
{-3}	0 ≤ addr′`offset
{-4}	addr′`offset < max_linear_offset
{-5}	delta′ ≥ 0
{-6}	rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{1}	offset(addr′)/expt(2, min_page) ≥ 0
{2}	expt(2, min_page) × floor(offset(addr′)/expt(2, min_page)) ≥ 0
{3}	offset(addr′) − rem(expt(2, min_page))(offset(addr′)) ≥ 0
{4}	addr′`offset + delta′ < max_linear_offset

Rewriting using nnreal_div_posreal_is_nnreal, matching in *,
This completes the proof of `delta_memory_address.2.2.2`.
Q.E.D.

C.117.57 Linear_Memory_Properties.xlat_idx_same_page_address_TCC1

Terse proof for `xlat_idx_same_page_address_TCC1`.

`xlat_idx_same_page_address_TCC1:`

{1} ∀ (addr: Memory_Address_4G, lvl: Level, a: Memory_Address_4G): offset(a) ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_idx_same_page_address_TCC1`.
Q.E.D.

C.117.58 Linear_Memory_Properties.xlat_idx_same_page_address_TCC2

Terse proof for `xlat_idx_same_page_address_TCC2`.

`xlat_idx_same_page_address_TCC2:`

{1} ∀ (addr: Memory_Address_4G, δ: nat, lvl: Level,
base:
{a: Memory_Address_4G |
aligned?(bits_per_level[Physical_memory, pm_phy] + pe_size)
(offset(a))}):
rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) ⊃
0 ≤ (addr + δ)`offset ∧ (addr + $\delta$)`offset < max_linear_offset

Repeatedly Skolemizing and flattening,
Rewriting using delta_memory_address, matching in *,
Expanding the definition of +,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_same_page_address_TCC2`.
Q.E.D.

C.117.59 Linear_Memory_Properties.xlat_idx_same_page_address

Terse proof for `xlat_idx_same_page_address`.

`xlat_idx_same_page_address:`

{1} ∀ (addr: Memory_Address_4G, δ: nat, lvl: Level,
base:
{a: Memory_Address_4G | aligned?(bits_per_level + pe_size)(offset(a))}):
rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) ⊃
xlat_idx(lvl, base, addr) = xlat_idx(lvl, base, addr + δ)

Repeatedly Skolemizing and flattening,
Expanding the definition of xlat_idx,
Expanding the definition of cut_bits,
Installing automatic rewrites from: (bus_width! bits_per_level! max_level! min_page!)
Case splitting on bus_width - bits_per_level - bits_per_level * lvl!1 >= 0,
we get 2 subgoals:

`xlat_idx_same_page_address.1:`

{-1} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-2} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-3} $0 \leq \text{addr}'\text{`offset}$
{-4} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-5} $\text{delta}' \geq 0$
{-6} $\text{lvl}' < \text{max\_level}$
{-7} $\text{Mem?}(\text{base}'\text{`type\_of})$
{-8} $0 \leq \text{base}'\text{`offset}$
{-9} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$
{-10} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-11} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page})$

{1} $\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2, \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times \text{bits\_per\_level}$
$=$
$\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2, \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \text{expt}(2, -1 \times \text{bits}$

Case splitting on ndiv(offset(addr!1), expt(2, -1 * (bits_per_level * lvl!1) - bits_per_level + bus_width)) = ndiv(offset(addr!1 + delta!1), expt(2, -1 * (bits_per_level * lvl!1) - bits_per_level + bus_width)),

we get 3 subgoals:

`xlat_idx_same_page_address.1.1:`

{-1} $\text{ndiv}(\text{offset}(\text{addr}'),$
$\text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
$=$
$\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'),$
$\text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
{-2} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-3} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-4} $0 \leq \text{addr}'\text{`offset}$
{-5} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-6} $\text{delta}' \geq 0$
{-7} $\text{lvl}' < \text{max\_level}$
{-8} $\text{Mem?}(\text{base}'\text{`type\_of})$
{-9} $0 \leq \text{base}'\text{`offset}$
{-10} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$
{-11} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-12} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page})$

{1} $\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2, \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2, -1 \times \text{bits\_per\_level}$
$=$
$\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2, \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'), \text{expt}(2, -1 \times \text{bits}$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.1`.

xlat_idx_same_page_address.1.2:

{-1} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-2} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-3} $0 \leq \text{addr}'\text{`offset}$
{-4} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-5} $\text{delta}' \geq 0$
{-6} $\text{lvl}' < \text{max\_level}$
{-7} $\text{Mem?}(\text{base}'\text{`type\_of})$
{-8} $0 \leq \text{base}'\text{`offset}$
{-9} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$
{-10} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-11} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page})$

{1} $\text{ndiv}(\text{offset}(\text{addr}'),\ \text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
$=$
$\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'),\ \text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$

{2} $\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2,\ \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}'),\ \text{expt}(2,\ -1 \times \text{bits\_per\_level} \times \text{lvl}' - \text{bit}$
$=$
$\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2,\ \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'),\ \text{expt}(2,\ -1 \times \text{bits\_per\_level} \times$

Hiding formulas: 2,

Expanding the definition of +,

Using lemma ndiv_plus_mod,

we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1:`

{-1} divides(expt(2,
−1 × (bits_per_level × lvl′) − bits_per_level + bus_width),
offset(addr′) − rem(expt(2, −1 × (bits_per_level × lvl′) − bits_per_level + bus_width))(
⊃
ndiv(offset(addr′) − rem(expt(2, −1 × (bits_per_level × lvl′) − bits_per_level + bus_width))(offset
expt(2,
−1 × (bits_per_level × lvl′) − bits_per_level + bus_width))
=
ndiv(offset(addr′) − rem(expt(2, −1 × (bits_per_level × lvl′) − bits_per_level + bus_width))(offs
expt(2,
−1 × (bits_per_level × lvl′) − bits_per_level + bus_width))
{-2} bus_width − bits_per_level − bits_per_level × lvl′ ≥ 0
{-3} Mem?(addr′‘type_of)
{-4} 0 ≤ addr′‘offset
{-5} addr′‘offset < max_linear_offset
{-6} delta′ ≥ 0
{-7} lvl′ < max_level
{-8} Mem?(base′‘type_of)
{-9} 0 ≤ base′‘offset
{-10} base′‘offset < max_linear_offset
{-11} aligned?(bits_per_level + pe_size)(offset(base′))
{-12} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)

{1} ndiv(offset(addr′),
expt(2, −1 × (bits_per_level × lvl′) − bits_per_level + bus_width))
=
ndiv(addr′‘offset + delta′,
expt(2, −1 × (bits_per_level × lvl′) − bits_per_level + bus_width))

Using lemma ndiv_plus_mod,

we get 3 subgoals:

xlat_idx_same_page_address.1.2.1.1:

{-1} divides(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$, offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)

⊃

ndiv(offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)) +

expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

=

ndiv(offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)),

expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

{-2} divides(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$, offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)

⊃

ndiv(offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)) +

expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

=

ndiv(offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)),

expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

{-3} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$

{-4} Mem?(addr′`type_of)

{-5} $0 \leq$ addr′`offset

{-6} addr′`offset $<$ max_linear_offset

{-7} $\text{delta}' \geq 0$

{-8} $\text{lvl}' <$ max_level

{-9} Mem?(base′`type_of)

{-10} $0 \leq$ base′`offset

{-11} base′`offset $<$ max_linear_offset

{-12} aligned?(bits_per_level + pe_size)(offset(base′))

{-13} rem(expt(2, min_page))(offset(addr′)) + $\text{delta}' <$ expt(2, min_page)

{1} ndiv(offset(addr′), expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

=

ndiv(addr′`offset + delta′, expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma rem_def2,

we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1.1.1:`

{-1}	TRUE ≡ divides(expt(2, 22 − 10 × lvl′), offset(addr′) − rem(expt(2, 22 − 10 × lvl′))(offset(addr′)))
{-2}	22 − 10 × lvl′ ≥ 0
{-3}	Mem?(addr′‘type_of)
{-4}	0 ≤ addr′‘offset
{-5}	addr′‘offset < max_linear_offset
{-6}	delta′ ≥ 0
{-7}	lvl′ < 2
{-8}	Mem?(base′‘type_of)
{-9}	0 ≤ base′‘offset
{-10}	base′‘offset < max_linear_offset
{-11}	aligned?(10 + pe_size)(offset(base′))
{-12}	rem(expt(2, 12))(offset(addr′)) + delta′ < expt(2, 12)
{1}	divides(expt(2, 22 − 10 × lvl′), offset(addr′) − rem(expt(2, 22 − 10 × lvl′))(offset(addr′)))
{2}	ndiv(offset(addr′), expt(2, 22 − 10 × lvl′)) = ndiv(addr′‘offset + delta′, expt(2, 22 − 10 × lvl′))

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `xlat_idx_same_page_address.1.2.1.1.1`.

`xlat_idx_same_page_address.1.2.1.1.2:`

{-1}	22 − 10 × lvl′ ≥ 0
{-2}	Mem?(addr′‘type_of)
{-3}	0 ≤ addr′‘offset
{-4}	addr′‘offset < max_linear_offset
{-5}	delta′ ≥ 0
{-6}	lvl′ < 2
{-7}	Mem?(base′‘type_of)
{-8}	0 ≤ base′‘offset
{-9}	base′‘offset < max_linear_offset
{-10}	aligned?(10 + pe_size)(offset(base′))
{-11}	rem(expt(2, 12))(offset(addr′)) + delta′ < expt(2, 12)
{1}	rem(expt(2, 22 − 10 × lvl′))(offset(addr′)) < expt(2, 22 − 10 × lvl′)
{2}	divides(expt(2, 22 − 10 × lvl′), offset(addr′) − rem(expt(2, 22 − 10 × lvl′))(offset(addr′)))
{3}	ndiv(offset(addr′), expt(2, 22 − 10 × lvl′)) = ndiv(addr′‘offset + delta′, expt(2, 22 − 10 × lvl′))

Adding type constraints for rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `xlat_idx_same_page_address.1.2.1.1.2`.

`xlat_idx_same_page_address.1.2.1.2:`

{-1} divides(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$, offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)
⊃
ndiv(offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)) +
expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)
=
ndiv(offset(addr′) − rem(expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)(offset(addr′)),
expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

{-2} bus_width − bits_per_level − bits_per_level × lvl′ ≥ 0
{-3} Mem?(addr′`type_of)
{-4} 0 ≤ addr′`offset
{-5} addr′`offset < max_linear_offset
{-6} delta′ ≥ 0
{-7} lvl′ < max_level
{-8} Mem?(base′`type_of)
{-9} 0 ≤ base′`offset
{-10} base′`offset < max_linear_offset
{-11} aligned?(bits_per_level + pe_size)(offset(base′))
{-12} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)

{1} rem(expt(2, $22 - 10 \times \text{lvl}'$))(offset(addr′)) + delta′ <
expt(2, $22 - 10 \times \text{lvl}'$)
{2} ndiv(offset(addr′),
expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)
=
ndiv(addr′`offset + delta′,
expt(2, $-1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width})$)

Keeping (-2 -6 -7 -11 -12 1) and hiding *,
Using lemma aligned_add_below,
we get 3 subgoals:

`xlat_idx_same_page_address.1.2.1.2.1:`

{-1} min_page ≤ bus_width − bits_per_level − bits_per_level × lvl′ ∧
rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))(offset(addr′)) − rem(expt(2, min_page))(offse
< expt(2, bus_width − bits_per_level − bits_per_level × lvl′)
∧
aligned?(min_page)
(rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))(offset(addr′)) − rem(expt(2, min_
∧ rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
⊃
rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))(offset(addr′)) − rem(expt(2, min_page))(offse
< expt(2, bus_width − bits_per_level − bits_per_level × lvl′)
{-2} bus_width − bits_per_level − bits_per_level × lvl′ ≥ 0
{-3} delta′ ≥ 0
{-4} lvl′ < max_level
{-5} aligned?(bits_per_level + pe_size)(offset(base′))
{-6} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)

{1} rem(expt(2, $22 - 10 \times \text{lvl}'$))(offset(addr′)) + delta′ <
expt(2, $22 - 10 \times \text{lvl}'$)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`xlat_idx_same_page_address.1.2.1.2.1.1:`

{-1} $\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{-2} $22 - 10 \times \text{lvl}' \geq 0$
{-3} $\text{delta}' \geq 0$
{-4} $\text{lvl}' < 2$
{-5} $\text{aligned?}(10 + \text{pe\_size})(\text{offset}(\text{base}'))$

{1} $\text{aligned?}(12)$
$(\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')))$
{2} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
$\text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Expanding the definition of aligned?,
Using lemma rem_def2,
Rewriting using rem_rem, matching in *,
we get 2 subgoals:
`xlat_idx_same_page_address.1.2.1.2.1.1.1:`

{-1} TRUE $\equiv$
$\text{divides}(\text{expt}(2,\ \text{min\_page}),$
$\text{rem}(\text{expt}(2,\ \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{ex}$
{-2} $\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{-3} $22 - 10 \times \text{lvl}' \geq 0$
{-4} $\text{delta}' \geq 0$
{-5} $\text{lvl}' < 2$
{-6} $\text{divides}(\text{expt}(2,\ 10 + \text{pe\_size}),\ \text{offset}(\text{base}'))$

{1} $\text{divides}(\text{expt}(2,\ 12),$
$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')))$
{2} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
$\text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_same_page_address.1.2.1.2.1.1.1`.
`xlat_idx_same_page_address.1.2.1.2.1.1.2:`

{-1} $\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{-2} $22 - 10 \times \text{lvl}' \geq 0$
{-3} $\text{delta}' \geq 0$
{-4} $\text{lvl}' < 2$
{-5} $\text{divides}(\text{expt}(2,\ 10 + \text{pe\_size}),\ \text{offset}(\text{base}'))$

{1} $\text{divides}(\text{expt}(2,\ 12),\ \text{expt}(2,\ 22 - 10 \times \text{lvl}'))$
{2} $\text{rem}(\text{expt}(2,\ 12))(\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}'))) =$
$\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}'))$
{3} $\text{divides}(\text{expt}(2,\ 12),$
$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')))$
{4} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
$\text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Keeping (-2 1) and hiding *,
Using lemma divides_expt_gt,
Rewriting using divides_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_same_page_address.1.2.1.2.1.1.2`.

`xlat_idx_same_page_address.1.2.1.2.1.2:`

{-1} $\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{-2} $22 - 10 \times \text{lvl}' \geq 0$
{-3} $\text{delta}' \geq 0$
{-4} $\text{lvl}' < 2$
{-5} $\text{aligned?}(10 + \text{pe\_size})(\text{offset}(\text{base}'))$

{1} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}'))$
$< \text{expt}(2,\ 22 - 10 \times \text{lvl}')$
{2} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
$\text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Adding type constraints for rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.2.1.2.1.2`.

`xlat_idx_same_page_address.1.2.1.2.2:`

{-1} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-2} $\text{delta}' \geq 0$
{-3} $\text{lvl}' < \text{max\_level}$
{-4} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-5} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}'))$
≥ 0
{2} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
$\text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Using lemma rem_floor,

we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1.2.2.1:`

{-1} $\text{rem}(\text{expt}(2,\ \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'))$
$(\text{offset}(\text{addr}'))$
$=$
$\text{rem}(\text{expt}(2,\ 12))(\text{rem}(\text{expt}(2,\ \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'))(\text{offset}(\text{addr}'))) + \text{expt}(2,\ 1$
{-2} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-3} $\text{delta}' \geq 0$
{-4} $\text{lvl}' < \text{max\_level}$
{-5} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-6} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}'))$
≥ 0
{2} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' <$
$\text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Rewriting using rem_rem, matching in *,

we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1.2.2.1.1:`

{-1} rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))
(offset(addr′))
=
rem(expt(2, 12))(offset(addr′)) + expt(2, 12) × floor(rem(expt(2, bus_width − bits_per_level − b
{-2} bus_width − bits_per_level − bits_per_level × lvl′ ≥ 0
{-3} delta′ ≥ 0
{-4} lvl′ < max_level
{-5} aligned?(bits_per_level + pe_size)(offset(base′))
{-6} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)

{1} rem(expt(2, 22 − 10 × lvl′))(offset(addr′)) − rem(expt(2, 12))(offset(addr′))
≥ 0
{2} rem(expt(2, 22 − 10 × lvl′))(offset(addr′)) + delta′ <
expt(2, 22 − 10 × lvl′)

Adding type constraints for expt(2, 12) * floor(rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)) / expt(2, min_page)),

we get 2 subgoals:

`xlat_idx_same_page_address.1.2.1.2.2.1.1.1:`

{-1} expt(2, 12) × floor(rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))(offset(addr′))/e
≥ 0
{-2} divides(expt(2, 12))
(expt(2, 12) × floor(rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))(offset(
{-3} divides(floor(rem(expt(2, −1 × bits_per_level × lvl′ − bits_per_level + bus_width))(offset(addr′))/ex
(expt(2, 12) × floor(rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))(offset(
{-4} rem(expt(2, bus_width − bits_per_level − bits_per_level × lvl′))
(offset(addr′))
=
rem(expt(2, 12))(offset(addr′)) + expt(2, 12) × floor(rem(expt(2, bus_width − bits_per_level − b
{-5} bus_width − bits_per_level − bits_per_level × lvl′ ≥ 0
{-6} delta′ ≥ 0
{-7} lvl′ < max_level
{-8} aligned?(bits_per_level + pe_size)(offset(base′))
{-9} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)

{1} rem(expt(2, 22 − 10 × lvl′))(offset(addr′)) − rem(expt(2, 12))(offset(addr′))
≥ 0
{2} rem(expt(2, 22 − 10 × lvl′))(offset(addr′)) + delta′ <
expt(2, 22 − 10 × lvl′)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.2.1.2.2.1.1.1`.

`xlat_idx_same_page_address.1.2.1.2.2.1.1.2:`

{-1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) = \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{expt}(2,\ 12) \times \mathrm{floor}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level}$
{-2} $\mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}' \geq 0$
{-3} $\mathrm{delta}' \geq 0$
{-4} $\mathrm{lvl}' < \mathrm{max\_level}$
{-5} $\mathrm{aligned?}(\mathrm{bits\_per\_level} + \mathrm{pe\_size})(\mathrm{offset}(\mathrm{base}'))$
{-6} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{min\_page}))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ \mathrm{min\_page})$

{1} $22 - 10 \times \mathrm{lvl}' \geq 0$
{2} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) - \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) \geq 0$
{3} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_same_page_address.1.2.1.2.2.1.1.2`.

`xlat_idx_same_page_address.1.2.1.2.2.1.2:`

{-1} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) = \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}'))) + \mathrm{expt}(2,\ 12) \times \mathrm{floor}(\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lv}$
{-2} $22 - 10 \times \mathrm{lvl}' \geq 0$
{-3} $\mathrm{delta}' \geq 0$
{-4} $\mathrm{lvl}' < 2$
{-5} $\mathrm{aligned?}(10 + \mathrm{pe\_size})(\mathrm{offset}(\mathrm{base}'))$
{-6} $\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$

{1} $\mathrm{divides}(\mathrm{expt}(2,\ 12),\ \mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))$
{2} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) - \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) \geq 0$
{3} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}')$

Keeping (-2 1) and hiding *,
Using lemma divides_expt_gt,
Rewriting using divides_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_same_page_address.1.2.1.2.2.1.2`.

`xlat_idx_same_page_address.1.2.1.2.2.2:`

{-1} $\mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}' \geq 0$
{-2} $\mathrm{delta}' \geq 0$
{-3} $\mathrm{lvl}' < \mathrm{max\_level}$
{-4} $\mathrm{aligned?}(\mathrm{bits\_per\_level} + \mathrm{pe\_size})(\mathrm{offset}(\mathrm{base}'))$
{-5} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{min\_page}))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ \mathrm{min\_page})$

{1} $22 - 10 \times \mathrm{lvl}' \geq 0$
{2} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) - \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) \geq 0$
{3} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_idx_same_page_address.1.2.1.2.2.2`.

`xlat_idx_same_page_address.1.2.1.2.3:`

{-1} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-2} $\text{delta}' \geq 0$
{-3} $\text{lvl}' < \text{max\_level}$
{-4} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-5} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $22 - 10 \times \text{lvl}' \geq 0$
{2} $\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 22 - 10 \times \text{lvl}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_idx_same_page_address.1.2.1.2.3`.

`xlat_idx_same_page_address.1.2.1.3:`

{-1} $\text{divides}(\text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}),$
$\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))(\text{c}$
$\supset$
$\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))(\text{offset}$
$\text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
$=$
$\text{ndiv}(\text{offset}(\text{addr}') - \text{rem}(\text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))(\text{offs}$
$\text{expt}(2, -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
{-2} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-3} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-4} $0 \leq \text{addr}'\text{`offset}$
{-5} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-6} $\text{delta}' \geq 0$
{-7} $\text{lvl}' < \text{max\_level}$
{-8} $\text{Mem?}(\text{base}'\text{`type\_of})$
{-9} $0 \leq \text{base}'\text{`offset}$
{-10} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$
{-11} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-12} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $22 - 10 \times \text{lvl}' \geq 0$
{2} $\text{ndiv}(\text{offset}(\text{addr}'), \text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
$=$
$\text{ndiv}(\text{addr}'\text{`offset} + \text{delta}', \text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `xlat_idx_same_page_address.1.2.1.3`.

`xlat_idx_same_page_address.1.2.2:`

{-1} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-2} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-3} $0 \leq \text{addr}'\text{`offset}$
{-4} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-5} $\text{delta}' \geq 0$
{-6} $\text{lvl}' < \text{max\_level}$
{-7} $\text{Mem?}(\text{base}'\text{`type\_of})$
{-8} $0 \leq \text{base}'\text{`offset}$
{-9} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$
{-10} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-11} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $22 - 10 \times \text{lvl}' \geq 0$
{2} $\text{ndiv}(\text{offset}(\text{addr}'),$
$\quad \text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$
$=$
$\text{ndiv}(\text{addr}'\text{`offset} + \text{delta}',$
$\quad \text{expt}(2,\ -1 \times (\text{bits\_per\_level} \times \text{lvl}') - \text{bits\_per\_level} + \text{bus\_width}))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `xlat_idx_same_page_address.1.2.2`.

`xlat_idx_same_page_address.1.3:`

{-1} $\text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \geq 0$
{-2} $\text{Mem?}(\text{addr}'\text{`type\_of})$
{-3} $0 \leq \text{addr}'\text{`offset}$
{-4} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$
{-5} $\text{delta}' \geq 0$
{-6} $\text{lvl}' < \text{max\_level}$
{-7} $\text{Mem?}(\text{base}'\text{`type\_of})$
{-8} $0 \leq \text{base}'\text{`offset}$
{-9} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$
{-10} $\text{aligned?}(\text{bits\_per\_level} + \text{pe\_size})(\text{offset}(\text{base}'))$
{-11} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page})$

{1} $22 - 10 \times \text{lvl}' \geq 0$
{2} $\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2,\ \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}'),\ \text{expt}(2,\ -1 \times \text{bits\_per\_level} \times \text{lvl}' - \text{bit}$
$=$
$\text{base}' + \text{shift\_bits\_left}(\text{rem}(\text{expt}(2,\ \text{bits\_per\_level}))(\text{ndiv}(\text{offset}(\text{addr}' + \text{delta}'),\ \text{expt}(2,\ -1 \times \text{bits\_per\_level} \times$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `xlat_idx_same_page_address.1.3`.

`xlat_idx_same_page_address.2:`

```
{-1}   Mem?(addr'`type_of)
{-2}   0 ≤ addr'`offset
{-3}   addr'`offset < max_linear_offset
{-4}   delta' ≥ 0
{-5}   lvl' < max_level
{-6}   Mem?(base'`type_of)
{-7}   0 ≤ base'`offset
{-8}   base'`offset < max_linear_offset
{-9}   aligned?(bits_per_level + pe_size)(offset(base'))
{-10}  rem(expt(2, min_page))(offset(addr')) + delta' < expt(2, min_page)
-------------------------------------------------------------------------------
{1}    bus_width − bits_per_level − bits_per_level × lvl' ≥ 0
{2}    base' + shift_bits_left(rem(expt(2, bits_per_level))(ndiv(offset(addr'), expt(2, −1 × bits_per_leve
       =
       base' + shift_bits_left(rem(expt(2, bits_per_level))(ndiv(offset(addr' + delta'), expt(2, −1 × bits
```

Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_idx_same_page_address.2`.
Q.E.D.

C.117.60 Linear_Memory_Properties.xlat_ofs_same_page_address_TCC1

Terse proof for `xlat_ofs_same_page_address_TCC1`.

`xlat_ofs_same_page_address_TCC1:`

```
{1}    ∀ (addr: Memory_Address_4G, lvl: Level, a: Memory_Address_4G):
         bus_width − (lvl + 1) × bits_per_level[Physical_memory, pm_phy] ≥ 0
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: (bits_per_level! max_level! bus_width! pe_size!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address_TCC1`.
Q.E.D.

C.117.61 Linear_Memory_Properties.xlat_ofs_same_page_address_TCC2

Terse proof for `xlat_ofs_same_page_address_TCC2`.

`xlat_ofs_same_page_address_TCC2:`

```
{1}    ∀ (addr: Memory_Address_4G, δ: nat, lvl: Level,
           base:
             {a: Memory_Address_4G |
                   aligned?(bus_width − (lvl + 1) × bits_per_level [Physical_memory, pm_phy])
                     (offset(a))}):
         rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) ⊃
          0 ≤ (addr + δ)`offset ∧ (addr + δ)`offset < max_linear_offset
```

Repeatedly Skolemizing and flattening,

Rewriting using delta_memory_address, matching in *,

Expanding the definition of +,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address_TCC2`.

Q.E.D.

C.117.62 Linear_Memory_Properties.xlat_ofs_same_page_address

Terse proof for `xlat_ofs_same_page_address`.

`xlat_ofs_same_page_address:`

{1} $\forall$ (addr: Memory_Address_4G, δ: nat, lvl: Level,
base:
{a: Memory_Address_4G |
aligned?(bus_width $-$ (lvl $+$ 1) $\times$ bits_per_level)
(offset(a))}):
rem(expt(2, min_page))(offset(addr)) $+ \delta <$ expt(2, min_page) $\supset$
xlat_ofs(lvl, base, addr) $+ \delta =$ xlat_ofs(lvl, base, addr $+ \delta$)

Repeatedly Skolemizing and flattening,

Expanding the definition of xlat_ofs,

Expanding the definition of cut_bits,

Installing automatic rewrites from: (expt_x0_aux! ndiv_1! bits_per_level! min_page! bus_width!)

Simplifying, rewriting, and recording with decision procedures,

Case splitting on min_page <= bus_width - bits_per_level - bits_per_level * lvl!1,

we get 2 subgoals:

`xlat_ofs_same_page_address.1:`

{-1} min_page $\leq$ bus_width $-$ bits_per_level $-$ bits_per_level $\times$ lvl$'$
{-2} Mem?(addr$'$`type_of)
{-3} $0 \leq$ addr$'$`offset
{-4} addr$'$`offset $<$ max_linear_offset
{-5} delta$' \geq 0$
{-6} lvl$' <$ max_level
{-7} Mem?(base$'$`type_of)
{-8} $0 \leq$ base$'$`offset
{-9} base$'$`offset $<$ max_linear_offset
{-10} aligned?($22 - 10 \times$ lvl$'$)(offset(base$'$))
{-11} rem(expt(2, 12))(offset(addr$'$)) $+$ delta$' <$ expt(2, 12)

{1} base$'$ $+$ rem(expt(2, $22 - 10 \times$ lvl$'$))(offset(addr$'$)) $+$ delta$' =$
base$'$ $+$ rem(expt(2, $22 - 10 \times$ lvl$'$))(offset(addr$'$ $+$ delta$'$))

Case splitting on rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)) + delta!1 = rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1 + delta!1)),

we get 3 subgoals:

`xlat_ofs_same_page_address.1.1:`

{-1}	$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}'$ $=$ $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))$ $(\mathrm{offset}(\mathrm{addr}' + \mathrm{delta}'))$
{-2}	$\mathrm{min\_page} \leq \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'$
{-3}	$\mathrm{Mem?}(\mathrm{addr}'\text{`}\mathrm{type\_of})$
{-4}	$0 \leq \mathrm{addr}'\text{`}\mathrm{offset}$
{-5}	$\mathrm{addr}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-6}	$\mathrm{delta}' \geq 0$
{-7}	$\mathrm{lvl}' < \mathrm{max\_level}$
{-8}	$\mathrm{Mem?}(\mathrm{base}'\text{`}\mathrm{type\_of})$
{-9}	$0 \leq \mathrm{base}'\text{`}\mathrm{offset}$
{-10}	$\mathrm{base}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-11}	$\mathrm{aligned?}(22 - 10 \times \mathrm{lvl}')(\mathrm{offset}(\mathrm{base}'))$
{-12}	$\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$
{1}	$\mathrm{base}' + \mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' =$ $\mathrm{base}' + \mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}' + \mathrm{delta}'))$

Expanding the definition of +,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.1`.

`xlat_ofs_same_page_address.1.2:`

{-1}	$\mathrm{min\_page} \leq \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'$
{-2}	$\mathrm{Mem?}(\mathrm{addr}'\text{`}\mathrm{type\_of})$
{-3}	$0 \leq \mathrm{addr}'\text{`}\mathrm{offset}$
{-4}	$\mathrm{addr}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-5}	$\mathrm{delta}' \geq 0$
{-6}	$\mathrm{lvl}' < \mathrm{max\_level}$
{-7}	$\mathrm{Mem?}(\mathrm{base}'\text{`}\mathrm{type\_of})$
{-8}	$0 \leq \mathrm{base}'\text{`}\mathrm{offset}$
{-9}	$\mathrm{base}'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-10}	$\mathrm{aligned?}(22 - 10 \times \mathrm{lvl}')(\mathrm{offset}(\mathrm{base}'))$
{-11}	$\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$
{1}	$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}'$ $=$ $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))$ $(\mathrm{offset}(\mathrm{addr}' + \mathrm{delta}'))$
{2}	$\mathrm{base}' + \mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' =$ $\mathrm{base}' + \mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}' + \mathrm{delta}'))$

Hiding formulas: 2,

Expanding the definition of +,

Using lemma rem_sum_assoc,

we get 2 subgoals:

`xlat_ofs_same_page_address.1.2.1:`

{-1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))$
$\quad(\mathrm{addr}'\text{`offset} + \mathrm{delta}')$
$=$
$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))(\mathrm{addr}'\text{`offset}) + \mathrm{delta}'$
$\equiv$
$\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))$
$\quad(\mathrm{addr}'\text{`offset})$
$< \mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}') - \mathrm{delta}'$

{-2} $\mathrm{min\_page} \leq \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'$

{-3} $\mathrm{Mem?}(\mathrm{addr}'\text{`type\_of})$

{-4} $0 \leq \mathrm{addr}'\text{`offset}$

{-5} $\mathrm{addr}'\text{`offset} < \mathrm{max\_linear\_offset}$

{-6} $\mathrm{delta}' \geq 0$

{-7} $\mathrm{lvl}' < \mathrm{max\_level}$

{-8} $\mathrm{Mem?}(\mathrm{base}'\text{`type\_of})$

{-9} $0 \leq \mathrm{base}'\text{`offset}$

{-10} $\mathrm{base}'\text{`offset} < \mathrm{max\_linear\_offset}$

{-11} $\mathrm{aligned?}(22 - 10 \times \mathrm{lvl}')(\mathrm{offset}(\mathrm{base}'))$

{-12} $\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$

{1} $\mathrm{rem}(\mathrm{expt}(2,\ -1 \times \mathrm{bits\_per\_level} \times \mathrm{lvl}' - \mathrm{bits\_per\_level} + \mathrm{bus\_width}))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}'$
$=$
$\mathrm{rem}(\mathrm{expt}(2,\ -1 \times \mathrm{bits\_per\_level} \times \mathrm{lvl}' - \mathrm{bits\_per\_level} + \mathrm{bus\_width}))$
$\quad(\mathrm{addr}'\text{`offset} + \mathrm{delta}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma aligned_add_below,

we get 2 subgoals:

`xlat_ofs_same_page_address.1.2.1.1:`

{-1} $\text{min\_page} \leq \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}' \wedge$
$\text{rem}(\text{expt}(2, \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min}$
$< \text{expt}(2, \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}')$
$\wedge$
$\text{aligned?}(\text{min\_page})$
$(\text{rem}(\text{expt}(2, \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}($
$\wedge \text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page})$
$\supset$
$\text{rem}(\text{expt}(2, \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, \text{min}$
$< \text{expt}(2, \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}')$

{-2} $12 \leq 22 - 10 \times \text{lvl}'$

{-3} $\text{Mem?}(\text{addr}'\text{`type\_of})$

{-4} $0 \leq \text{addr}'\text{`offset}$

{-5} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$

{-6} $\text{delta}' \geq 0$

{-7} $\text{lvl}' < \text{max\_level}$

{-8} $\text{Mem?}(\text{base}'\text{`type\_of})$

{-9} $0 \leq \text{base}'\text{`offset}$

{-10} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$

{-11} $\text{aligned?}(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$

{-12} $\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$

{1} $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset} + \text{delta}') =$
$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset}) + \text{delta}'$

{2} $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset}) <$
$\text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$

{3} $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' =$
$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset} + \text{delta}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting, we get 2 subgoals:

`xlat_ofs_same_page_address.1.2.1.1.1:`

{-1} $12 \leq 22 - 10 \times \text{lvl}'$

{-2} $\text{Mem?}(\text{addr}'\text{`type\_of})$

{-3} $0 \leq \text{addr}'\text{`offset}$

{-4} $\text{addr}'\text{`offset} < \text{max\_linear\_offset}$

{-5} $\text{delta}' \geq 0$

{-6} $\text{lvl}' < \text{max\_level}$

{-7} $\text{Mem?}(\text{base}'\text{`type\_of})$

{-8} $0 \leq \text{base}'\text{`offset}$

{-9} $\text{base}'\text{`offset} < \text{max\_linear\_offset}$

{-10} $\text{aligned?}(22 - 10 \times \text{lvl}')(\text{offset}(\text{base}'))$

{-11} $\text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, 12)$

{1} $\text{aligned?}(12)$
$(\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2, 12))(\text{offset}(\text{addr}')))$

{2} $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset} + \text{delta}') =$
$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset}) + \text{delta}'$

{3} $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset}) <$
$\text{expt}(2, 22 - 10 \times \text{lvl}') - \text{delta}'$

{4} $\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' =$
$\text{rem}(\text{expt}(2, 22 - 10 \times \text{lvl}'))(\text{addr}'\text{`offset} + \text{delta}')$

Expanding the definition of aligned?,

Using lemma rem_def2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using rem_rem, matching in *,
Using lemma divides_expt_gt,
Rewriting using divides_reflexive, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address.1.2.1.1.1`.
`xlat_ofs_same_page_address.1.2.1.1.2`:

{-1}	$12 \leq 22 - 10 \times \text{lvl}'$
{-2}	Mem?(addr′‘type_of)
{-3}	$0 \leq$ addr′‘offset
{-4}	addr′‘offset $<$ max_linear_offset
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' <$ max_level
{-7}	Mem?(base′‘type_of)
{-8}	$0 \leq$ base′‘offset
{-9}	base′‘offset $<$ max_linear_offset
{-10}	aligned?$(22 - 10 \times \text{lvl}')$(offset(base′))
{-11}	$\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{1}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) < \text{expt}(2,\ 22 - 10 \times \text{lvl}')$
{2}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset} + \text{delta}') = \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset}) + \text{delta}'$
{3}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset}) < \text{expt}(2,\ 22 - 10 \times \text{lvl}') - \text{delta}'$
{4}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' = \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset} + \text{delta}')$

Adding type constraints for rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address.1.2.1.1.2`.
`xlat_ofs_same_page_address.1.2.1.2`:

{-1}	$12 \leq 22 - 10 \times \text{lvl}'$
{-2}	Mem?(addr′‘type_of)
{-3}	$0 \leq$ addr′‘offset
{-4}	addr′‘offset $<$ max_linear_offset
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' <$ max_level
{-7}	Mem?(base′‘type_of)
{-8}	$0 \leq$ base′‘offset
{-9}	base′‘offset $<$ max_linear_offset
{-10}	aligned?$(22 - 10 \times \text{lvl}')$(offset(base′))
{-11}	$\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{1}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) - \text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) \geq 0$
{2}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset} + \text{delta}') = \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset}) + \text{delta}'$
{3}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset}) < \text{expt}(2,\ 22 - 10 \times \text{lvl}') - \text{delta}'$
{4}	$\text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' = \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{addr}'\text{‘offset} + \text{delta}')$

Using lemma rem_floor,

Rewriting using rem_rem, matching in *,

we get 2 subgoals:

`xlat_ofs_same_page_address.1.2.1.2.1:`

{-1} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) = \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{expt}(2,\ 12) \times \mathrm{floor}(\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{bus\_width} - \mathrm{bits\_per\_level} -$

{-2} $12 \leq 22 - 10 \times \mathrm{lvl}'$

{-3} Mem?(addr′‘type_of)

{-4} $0 \leq$ addr′‘offset

{-5} addr′‘offset $<$ max_linear_offset

{-6} $\mathrm{delta}' \geq 0$

{-7} $\mathrm{lvl}' <$ max_level

{-8} Mem?(base′‘type_of)

{-9} $0 \leq$ base′‘offset

{-10} base′‘offset $<$ max_linear_offset

{-11} $\mathrm{aligned?}(22 - 10 \times \mathrm{lvl}')(\mathrm{offset}(\mathrm{base}'))$

{-12} $\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$

{1} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) - \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) \geq 0$

{2} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{‘}\mathrm{offset} + \mathrm{delta}') = \mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{‘}\mathrm{offset}) + \mathrm{delta}'$

{3} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{‘}\mathrm{offset}) < \mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}') - \mathrm{delta}'$

{4} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' = \mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{‘}\mathrm{offset} + \mathrm{delta}')$

Adding type constraints for expt(2, min_page) * floor(rem(expt(2, bus_width - bits_per_level - bits_per_level * lvl!1)) (offset(addr!1)) / expt(2, min_page)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.2.1.2.1`.

`xlat_ofs_same_page_address.1.2.1.2.2:`

{-1} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) =$
$\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}'))) + \mathrm{expt}(2,\ 12) \times \mathrm{floor}(\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times$
{-2} $12 \leq 22 - 10 \times \mathrm{lvl}'$
{-3} $\mathrm{Mem?}(\mathrm{addr}'\text{`type\_of})$
{-4} $0 \leq \mathrm{addr}'\text{`offset}$
{-5} $\mathrm{addr}'\text{`offset} < \mathrm{max\_linear\_offset}$
{-6} $\mathrm{delta}' \geq 0$
{-7} $\mathrm{lvl}' < \mathrm{max\_level}$
{-8} $\mathrm{Mem?}(\mathrm{base}'\text{`type\_of})$
{-9} $0 \leq \mathrm{base}'\text{`offset}$
{-10} $\mathrm{base}'\text{`offset} < \mathrm{max\_linear\_offset}$
{-11} $\mathrm{aligned?}(22 - 10 \times \mathrm{lvl}')(\mathrm{offset}(\mathrm{base}'))$
{-12} $\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$

{1} $\mathrm{divides}(\mathrm{expt}(2,\ 12),\ \mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))$
{2} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) - \mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}'))$
≥ 0
{3} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{`offset} + \mathrm{delta}') =$
$\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{`offset}) + \mathrm{delta}'$
{4} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{`offset}) <$
$\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}') - \mathrm{delta}'$
{5} $\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' =$
$\mathrm{rem}(\mathrm{expt}(2,\ 22 - 10 \times \mathrm{lvl}'))(\mathrm{addr}'\text{`offset} + \mathrm{delta}')$

Using lemma divides_expt_gt,

Rewriting using divides_reflexive, matching in *,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.2.1.2.2`.

`xlat_ofs_same_page_address.1.2.2:`

{-1} $\mathrm{min\_page} \leq \mathrm{bus\_width} - \mathrm{bits\_per\_level} - \mathrm{bits\_per\_level} \times \mathrm{lvl}'$
{-2} $\mathrm{Mem?}(\mathrm{addr}'\text{`type\_of})$
{-3} $0 \leq \mathrm{addr}'\text{`offset}$
{-4} $\mathrm{addr}'\text{`offset} < \mathrm{max\_linear\_offset}$
{-5} $\mathrm{delta}' \geq 0$
{-6} $\mathrm{lvl}' < \mathrm{max\_level}$
{-7} $\mathrm{Mem?}(\mathrm{base}'\text{`type\_of})$
{-8} $0 \leq \mathrm{base}'\text{`offset}$
{-9} $\mathrm{base}'\text{`offset} < \mathrm{max\_linear\_offset}$
{-10} $\mathrm{aligned?}(22 - 10 \times \mathrm{lvl}')(\mathrm{offset}(\mathrm{base}'))$
{-11} $\mathrm{rem}(\mathrm{expt}(2,\ 12))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}' < \mathrm{expt}(2,\ 12)$

{1} $22 - 10 \times \mathrm{lvl}' \geq 0$
{2} $\mathrm{rem}(\mathrm{expt}(2,\ -1 \times \mathrm{bits\_per\_level} \times \mathrm{lvl}' - \mathrm{bits\_per\_level} + \mathrm{bus\_width}))(\mathrm{offset}(\mathrm{addr}')) + \mathrm{delta}'$
$=$
$\mathrm{rem}(\mathrm{expt}(2,\ -1 \times \mathrm{bits\_per\_level} \times \mathrm{lvl}' - \mathrm{bits\_per\_level} + \mathrm{bus\_width}))$
$(\mathrm{addr}'\text{`offset} + \mathrm{delta}')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `xlat_ofs_same_page_address.1.2.2`.

`xlat_ofs_same_page_address.1.3:`

{-1}	$\text{min\_page} \leq \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'$
{-2}	Mem?(addr′‘type_of)
{-3}	$0 \leq$ addr′‘offset
{-4}	addr′‘offset $<$ max_linear_offset
{-5}	$\text{delta}' \geq 0$
{-6}	$\text{lvl}' <$ max_level
{-7}	Mem?(base′‘type_of)
{-8}	$0 \leq$ base′‘offset
{-9}	base′‘offset $<$ max_linear_offset
{-10}	aligned?$(22 - 10 \times \text{lvl}')$(offset(base′))
{-11}	$\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{1}	$22 - 10 \times \text{lvl}' \geq 0$
{2}	$\text{base}' + \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' =$ $\text{base}' + \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}' + \text{delta}'))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address.1.3`.
`xlat_ofs_same_page_address.2:`

{-1}	Mem?(addr′‘type_of)
{-2}	$0 \leq$ addr′‘offset
{-3}	addr′‘offset $<$ max_linear_offset
{-4}	$\text{delta}' \geq 0$
{-5}	$\text{lvl}' <$ max_level
{-6}	Mem?(base′‘type_of)
{-7}	$0 \leq$ base′‘offset
{-8}	base′‘offset $<$ max_linear_offset
{-9}	aligned?$(22 - 10 \times \text{lvl}')$(offset(base′))
{-10}	$\text{rem}(\text{expt}(2,\ 12))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ 12)$
{1}	$\text{min\_page} \leq \text{bus\_width} - \text{bits\_per\_level} - \text{bits\_per\_level} \times \text{lvl}'$
{2}	$\text{base}' + \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}')) + \text{delta}' =$ $\text{base}' + \text{rem}(\text{expt}(2,\ 22 - 10 \times \text{lvl}'))(\text{offset}(\text{addr}' + \text{delta}'))$

Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `xlat_ofs_same_page_address.2`.
Q.E.D.

C.117.63 Linear_Memory_Properties.translate_same_page_address_ok

Terse proof for `translate_same_page_address_ok`.
`translate_same_page_address_ok:`

{1}	$\forall$ (s: Linear_memory⌊Physical_memory, pm_phy⌋, addr: Memory_Address_4G, δ: nat, lvl: Level, base: {a: Memory_Address_4G | aligned?(bits_per_level + pe_size)(offset(a))}, access: Memory_access, priv: Memory_privilege): $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr})) + \delta < \text{expt}(2,\ \text{min\_page}) \wedge$ OK?(translate(lvl, base, addr, access, priv)(s)) $\supset$ OK?(translate(lvl, base, addr + δ, access, priv)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of translate,
Case splitting on xlat_idx(lvl!1, base!1, addr!1 + delta!1) = xlat_idx(lvl!1, base!1, addr!1),
we get 2 subgoals:

`translate_same_page_address_ok.1:`

```
{-1}  xlat_idx(lvl′, base′, addr′ + delta′) = xlat_idx(lvl′, base′, addr′)
{-2}  Mem?(addr′`type_of)
{-3}  0 ≤ addr′`offset
{-4}  addr′`offset < max_linear_offset
{-5}  delta′ ≥ 0
{-6}  lvl′ < max_level
{-7}  Mem?(base′`type_of)
{-8}  0 ≤ base′`offset
{-9}  base′`offset < max_linear_offset
{-10} aligned?(bits_per_level + pe_size)(offset(base′))
{-11} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{-12} OK?((read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm_phy, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
             (s′))
-------------------------------------------------------------------------------
{1}   OK?((read_data(pm_phy, paging_data_type(lvl′))
                     (xlat_idx(lvl′, base′, addr′ + delta′))
            ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm_phy, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′ + delta′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′ + delta′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′ + delta′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
             (s′))
```

Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res! raise_fault! excep-

tion_result! ok_result! fatal_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_same_page_address_ok.1`.

`translate_same_page_address_ok.2`:

```
{-1}  Mem?(addr′‘type_of)
{-2}  0 ≤ addr′‘offset
{-3}  addr′‘offset < max_linear_offset
{-4}  delta′ ≥ 0
{-5}  lvl′ < max_level
{-6}  Mem?(base′‘type_of)
{-7}  0 ≤ base′‘offset
{-8}  base′‘offset < max_linear_offset
{-9}  aligned?(bits_per_level + pe_size)(offset(base′))
{-10} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{-11} OK?((read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, addr′),
                                        set_reference(pe, access′))
                                      ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s′))
|-----
{1}   xlat_idx(lvl′, base′, addr′ + delta′) = xlat_idx(lvl′, base′, addr′)
{2}   OK?((read_data(pm_phy, paging_data_type(lvl′))
                     (xlat_idx(lvl′, base′, addr′ + delta′))
            ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm_phy, paging_data_type(lvl′))
                                      (xlat_idx(lvl′, base′, addr′ + delta′),
                                        set_reference(pe, access′))
                                      ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′ + delta′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′ + delta′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
          (s′))
```

Hiding formulas: (-9 2),

Using lemma xlat_idx_same_page_address,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_same_page_address_ok.2`.

Q.E.D.

C.117.64 Linear_Memory_Properties.translate_same_page_address_result

Terse proof for `translate_same_page_address_result`.

`translate_same_page_address_result`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, δ: nat, lvl: Level,
base: {a: Memory_Address_4G | aligned?(bits_per_level + pe_size)(offset(a))},
access: Memory_access, priv: Memory_privilege):
rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) ∧
OK?(translate(lvl, base, addr, access, priv)(s))
⊃
translate(lvl, base, addr, access, priv)(s) =
translate(lvl, base, addr + δ, access, priv)(s)

Repeatedly Skolemizing and flattening,

Using lemma translate_same_page_address_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of translate,

Case splitting on xlat_idx(lvl!1, base!1, addr!1 + delta!1) = xlat_idx(lvl!1, base!1, addr!1),

we get 2 subgoals:

`translate_same_page_address_result.1:`

```
{-1}  xlat_idx(lvl′, base′, addr′ + delta′) = xlat_idx(lvl′, base′, addr′)
{-2}  rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{-3}  OK?((read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm_phy, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
             (s′))
{-4}  OK?((read_data(pm_phy, paging_data_type(lvl′))
                    (xlat_idx(lvl′, base′, addr′ + delta′))
            ##
            (λ (pe: (range_pt(lvl′))):
               IF paging_type?(lvl′, pe)
                 THEN IF present?(pe)
                        THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                               THEN write_data(pm_phy, paging_data_type(lvl′))
                                               (xlat_idx(lvl′, base′, addr′ + delta′),
                                                 set_reference(pe, access′))
                                    ## ok_result(is_leaf?(pe), base(pe))
                             ELSE raise_fault(priv′, access′, TRUE, addr′ + delta′)
                             ENDIF
                      ELSE raise_fault(priv′, access′, FALSE, addr′ + delta′)
                      ENDIF
                 ELSE fatal_result
                 ENDIF))
             (s′))
{-5}  Mem?(addr′‘type_of)
{-6}  0 ≤ addr′‘offset
{-7}  addr′‘offset < max_linear_offset
{-8}  delta′ ≥ 0
{-9}  lvl′ < max_level
{-10} Mem?(base′‘type_of)
{-11} 0 ≤ base′‘offset
{-12} base′‘offset < max_linear_offset
{-13} aligned?(bits_per_level + pe_size)(offset(base′))
-------------------------------------------------------------------------
 {1}  (read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
         (λ (pe: (range_pt(lvl′))):
            IF paging_type?(lvl′, pe)
              THEN IF present?(pe)
                     THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
                            THEN write_data(pm_phy, paging_data_type(lvl′))
                                            (xlat_idx(lvl′, base′, addr′),
                                              set_reference(pe, access′))
                                 ## ok_result(is_leaf?(pe), base(pe))
                          ELSE raise_fault(priv′, access′, TRUE, addr′)
                          ENDIF
                   ELSE raise_fault(priv′, access′, FALSE, addr′)
                   ENDIF
              ELSE fatal_result
              ENDIF))
         (s′)
       =
       (read_data(pm_phy, paging_data_type(lvl′))
```

Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res! raise_fault! exception_result! fatal_result! ok_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `translate_same_page_address_result.1`.

translate_same_page_address_result.2:

{-1} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{-2} OK?((read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
(λ (pe: (range_pt(lvl′))):
IF paging_type?(lvl′, pe)
THEN IF present?(pe)
THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
THEN write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, addr′),
set_reference(pe, access′))
ok_result(is_leaf?(pe), base(pe))
ELSE raise_fault(priv′, access′, TRUE, addr′)
ENDIF
ELSE raise_fault(priv′, access′, FALSE, addr′)
ENDIF
ELSE fatal_result
ENDIF))
(s′))
{-3} OK?((read_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, addr′ + delta′))
##
(λ (pe: (range_pt(lvl′))):
IF paging_type?(lvl′, pe)
THEN IF present?(pe)
THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
THEN write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, addr′ + delta′),
set_reference(pe, access′))
ok_result(is_leaf?(pe), base(pe))
ELSE raise_fault(priv′, access′, TRUE, addr′ + delta′)
ENDIF
ELSE raise_fault(priv′, access′, FALSE, addr′ + delta′)
ENDIF
ELSE fatal_result
ENDIF))
(s′))
{-4} Mem?(addr′‘type_of)
{-5} 0 ≤ addr′‘offset
{-6} addr′‘offset < max_linear_offset
{-7} delta′ ≥ 0
{-8} lvl′ < max_level
{-9} Mem?(base′‘type_of)
{-10} 0 ≤ base′‘offset
{-11} base′‘offset < max_linear_offset
{-12} aligned?(bits_per_level + pe_size)(offset(base′))

{1} xlat_idx(lvl′, base′, addr′ + delta′) = xlat_idx(lvl′, base′, addr′)
{2} (read_data(pm_phy, paging_data_type(lvl′))(xlat_idx(lvl′, base′, addr′)) ##
(λ (pe: (range_pt(lvl′))):
IF paging_type?(lvl′, pe)
THEN IF present?(pe)
THEN IF accessible?(pe, access′) ∧ privileged?(pe, priv′)
THEN write_data(pm_phy, paging_data_type(lvl′))
(xlat_idx(lvl′, base′, addr′),
set_reference(pe, access′))
ok_result(is_leaf?(pe), base(pe))
ELSE raise_fault(priv′, access′, TRUE, addr′)
ENDIF
ELSE raise_fault(priv′, access′, FALSE, addr′)
ENDIF
ELSE fatal_result
ENDIF))
(s′)
=
(read_data(pm_phy, paging_data_type(lvl′))

Hiding formulas: (-2 -3 2),
Using lemma xlat_idx_same_page_address,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `translate_same_page_address_result.2`.
Q.E.D.

C.117.65 Linear_Memory_Properties.linear_resolve_same_page_address_ok_TCC1

Terse proof for `linear_resolve_same_page_address_ok_TCC1`.

`linear_resolve_same_page_address_ok_TCC1`:

{1} $\forall$ (s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, δ: nat,
ac: Memory_access):
rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) $\wedge$
OK?(linear_resolve(addr, ac)(s))
$\supset$ 0 $\leq$ (addr + δ)`offset $\wedge$ (addr + $\delta$)`offset < max_linear_offset

Repeatedly Skolemizing and flattening,
Rewriting using delta_memory_address, matching in *,
Expanding the definition of +,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_page_address_ok_TCC1`.
Q.E.D.

C.117.66 Linear_Memory_Properties.linear_resolve_same_page_address_ok

Terse proof for `linear_resolve_same_page_address_ok`.

`linear_resolve_same_page_address_ok`:

{1} $\forall$ (s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, δ: nat,
ac: Memory_access):
rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) $\wedge$
OK?(linear_resolve(addr, ac)(s))
$\supset$ OK?(linear_resolve(addr + δ, ac)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of linear_resolve,
Installing automatic rewrites from: (##! translate_same_page_address_ok! translate_same_page_address_result! min_linear! max_linear! Mem! Address_Helpers.<=! <! xlat_idx_memory_address! xlat_idx_memory_address2! ptab_lvl! pdir_lvl! max_level! expr_2_super! expr_2_super_res! ok_result! bus_width! bits_per_level! pe_size!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

```
linear_resolve_same_page_address_ok.1:

{-1}  Mem?(addr′‘type_of)
{-2}  0 ≤ addr′‘offset
{-3}  addr′‘offset < max_linear_offset
{-4}  delta′ ≥ 0
{-5}  rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{-6}  OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s′))
{-7}  OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
                    (state(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
{-8}  OK?(translate(0,
                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                   (s′))))‘base_addr,
                    addr′, ac′,
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                   (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                   (s′))))))
{-9}  data(translate(0,
                     data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s′))))‘base_addr,
                     addr′, ac′,
                     segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s′))))
                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                    (state(read_data(pm_phy, segment_reg_data_type)(CS)
                                                    (s′))))))‘1
{-10} OK?(OK(state(translate(0,
                             data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                            (state(read_data(pm_phy, segment_reg_data_type)
                                                            (CS)
                                                            (s′))))‘base_addr,
                             addr′, ac′,
                             segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                            (s′))))
                           (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                            (state(read_data(pm_phy, segment_reg_data_type)
                                                            (CS)
                                                            (s′))))),
                    xlat_ofs(0,
                             data(translate(0,
                                            data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                                           (state(read_data
                                                                  (pm_phy, segment_reg_data_type)
                                                                  (CS)
                                                                  (s′))))‘base_addr,
                                            addr′, ac′,
                                            segment_to_priv(data(read_data(pm_phy,
                                                                           segment_reg_data_type)
                                                                           (CS)
                                                                           (s′))))
                                          (state(read_data(pm_phy, pdbr_data_type)(PDBR)
```

Using lemma translate_same_page_address_ok,

Using lemma translate_same_page_address_result,

Expanding the definition of linear_resolve,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.1`.

linear_resolve_same_page_address_ok.2:

{-1} Mem?(addr′‘type_of)
{-2} 0 ≤ addr′‘offset
{-3} addr′‘offset < max_linear_offset
{-4} delta′ ≥ 0
{-5} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page)
{-6} OK?(read_data(pm_phy, segment_reg_data_type)(CS)(s′))
{-7} OK?(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
{-8} OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
{-9} OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))))‘2,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))‘base_addr
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
seg-
ment_reg_data_type)
(CS)
(s′))))))))
{-10} OK?(OK(state(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(P
(state
(read_data
(pm_phy, seg-

Using lemma translate_same_page_address_result,

Using lemma translate_same_page_address_ok,

Expanding the definition of linear_resolve,

Case splitting on 0 <= (addr!1 + delta!1)`offset, (addr!1 + delta!1)`offset < max_linear_offset,

we get 3 subgoals:

`linear_resolve_same_page_address_ok.2.1:`

{-1} $(\text{addr}' + \text{delta}')\text{`offset} < \text{max\_linear\_offset}$

{-2} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-3} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$

{-4} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s')))))$
$=$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})(s'))))$

Case splitting on OK?(translate(1, data(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))‘base_addr, addr!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))))‘2, addr!1, ac!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))‘base_addr, addr!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))))))) = OK?(translate(1, data(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))‘base_addr, addr!1 + delta!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))))‘2, addr!1 + delta!1, ac!1, segment_to_priv(data(read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(translate(0, data(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1))))‘base_addr, addr!1 + delta!1, ac!1, segment_to_priv(data (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))) (state(read_data (pm_phy, pdbr_data_type) (PDBR) (state (read_data (pm_phy, segment_reg_data_type) (CS) (s!1)))))))),

we get 7 subgoals:

```
linear_resolve_same_page_address_ok.2.1.1:

{-1}  OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, segment_reg_data_type)
                                               (CS)
                                               (s'))))`base_addr,
                                   addr', ac',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                        (CS)
                                                        (s'))))
                                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)
                                                (CS)
                                                (s'))))))`2,
                    addr', ac',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s'))))
                   (state(translate(0,
                                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy, segment_reg_data_type)
                                                (CS)
                                                (s'))))`base_addr
                                    addr', ac',
                                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                         (CS)
                                                         (s'))))
                                   (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                          (state(read_data(pm_phy,
                                                           segment_reg_data_type)
                                                 (CS)
                                                 (s'))))))))
      =
      OK?(translate(1,
                    data(translate(0,
                                   data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                        (state(read_data(pm_phy, segment_reg_data_type)
                                               (CS)
                                               (s'))))`base_addr
                                   addr' + delta', ac',
                                   segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
                                                        (CS)
                                                        (s'))))
                                  (state(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy,
                                                          segment_reg_data_type)
                                                (CS)
                                                (s'))))))`2,
                    addr' + delta', ac',
                    segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
                                                  (s'))))
                   (state(translate(0,
                                    data(read_data(pm_phy, pdbr_data_type)(PDBR)
                                         (state(read_data(pm_phy,
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.1`.

linear_resolve_same_page_address_ok.2.1.2:

{-1} (addr′ + delta′)‘offset < max_linear_offset
{-2} 0 ≤ (addr′ + delta′)‘offset
{-3} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
{-4} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′)))))
=
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))

Hiding formulas: 3,

Using lemma translate_same_page_address_ok,

we get 2 subgoals:

linear_resolve_same_page_address_ok.2.1.2.1:

{-1} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))‘base_addr
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s′))))))‘2,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s′))))‘base_add
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data
(pm_phy, segment_reg_data_type)
(CS)
(s′))))))))
⊃
OK?(translate(1,
data(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))‘base_addr
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)
(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy,
segment_reg_data_type)
(CS)
(s′))))))‘2,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

which is trivially true.

This completes the proof of `linear_resolve_same_page_address_ok.2.1.2.1`.

linear_resolve_same_page_address_ok.2.1.2.2:

{-1} $(\text{addr}' + \text{delta}')\text{`offset} < \text{max\_linear\_offset}$

{-2} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-3} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$

{-4} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s')))))$
$=$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s'))))$

Hiding formulas: (-13 -14 2),

Using lemma translate_result_no_leaf,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -7,

which is trivially true.

This completes the proof of `linear_resolve_same_page_address_ok.2.1.2.2`.

linear_resolve_same_page_address_ok.2.1.3:

{-1} (addr′ + delta′)‘offset < max_linear_offset
{-2} 0 ≤ (addr′ + delta′)‘offset
{-3} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
{-4} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′)))))
2058 =
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))

Using lemma translate_memory_address,

we get 2 subgoals:

`linear_resolve_same_page_address_ok.2.1.3.1:`

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr))
∧
OK?(translate(0,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))))
⊃
data(translate(0,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))))‘2
< max_linear
{-2} (addr′ + delta′)‘offset < max_linear_offset
{-3} 0 ≤ (addr′ + delta′)‘offset
{-4} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, seg-

Using lemma translate_memory_address2,

we get 2 subgoals:

```
linear_resolve_same_page_address_ok.2.1.3.1.1:
```

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr))
∧
OK?(translate(0,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))))
⊃
0 ≤
offset
(data(translate(0,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))))‘2)

{-2} aligned?(bits_per_level + pe_size)
(offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.3.1.1`.

linear_resolve_same_page_address_ok.2.1.3.1.2:

{-1} aligned?(bits_per_level + pe_size)
(offset
(data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr))
∧
OK?(translate(0,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))))
⊃
data(translate(0,
data[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))
(state[Physical_memory, Pdbr_type]
(read_data[Physical_memory, Pdbr_type]
(pm_phy, pdbr_data_type)(PDBR)
(state[Physical_memory, Segment_Reg_type]
(read_data[Physical_memory, Segment_Reg_type]
(pm_phy, segment_reg_data_type)(CS)(s′))))))‘2
< max_linear
{-2} (addr′ + delta′)‘offset < max_linear_offset
{-3} 0 ≤ (addr′ + delta′)‘offset
{-4} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, seg-

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.3.1.2`.

`linear_resolve_same_page_address_ok.2.1.3.2:`

{-1} $(\text{addr}' + \text{delta}')\text{`offset} < \text{max\_linear\_offset}$

{-2} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-3}
$$\begin{aligned}
&\text{rem}(\text{expt}(2,\ \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min_page}) \wedge \\
&\ \text{OK?}(\text{translate}(0,\\
&\qquad \text{data}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))\text{`base_addr},\\
&\qquad \text{addr}',\ \text{ac}',\\
&\qquad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad (s'))))\\
&\qquad\ (\text{state}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))))\\
&\ \supset\\
&\ \text{OK?}(\text{translate}(0,\\
&\qquad \text{data}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))\text{`base_addr},\\
&\qquad \text{addr}' + \text{delta}',\ \text{ac}',\\
&\qquad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad (s'))))\\
&\qquad\ (\text{state}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))))
\end{aligned}$$

{-4}
$$\begin{aligned}
&\text{rem}(\text{expt}(2,\ \text{min_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min_page}) \wedge \\
&\ \text{OK?}(\text{translate}(0,\\
&\qquad \text{data}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))\text{`base_addr},\\
&\qquad \text{addr}',\ \text{ac}',\\
&\qquad \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad (s'))))\\
&\qquad\ (\text{state}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))))\\
&\ \supset\\
&\ \text{translate}(0,\\
&\qquad\ \text{data}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))\text{`base_addr},\\
&\qquad\ \text{addr}',\ \text{ac}',\\
&\qquad\ \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})(s'))))\\
&\qquad\ (\text{state}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s')))))\\
&\ =\\
&\ \text{translate}(0,\\
&\qquad\ \text{data}(\text{read_data}(\text{pm_phy},\ \text{pdbr_data_type})(\text{PDBR})\\
&\qquad\qquad (\text{state}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})\\
&\qquad\qquad\qquad (s'))))\text{`base_addr},\\
&\qquad\ \text{addr}' + \text{delta}',\ \text{ac}',\\
&\qquad\ \text{segment_to_priv}(\text{data}(\text{read_data}(\text{pm_phy},\ \text{segment_reg_data_type})(\text{CS})(s'))))
\end{aligned}$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.3.2`.

linear_resolve_same_page_address_ok.2.1.4:

{-1} $(\text{addr}' + \text{delta}')\text{`offset} < \text{max\_linear\_offset}$

{-2} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-3} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$

{-4} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s')))))$
$=$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s'))))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.4`.

linear_resolve_same_page_address_ok.2.1.5:

{-1} $(\text{addr}' + \text{delta}')\text{`offset} < \text{max\_linear\_offset}$

{-2} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-3} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$

{-4} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2, \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s')))))$
$=$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy}, \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}', \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy}, \text{segment\_reg\_data\_type})(\text{CS})(s'))))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.5`.

linear_resolve_same_page_address_ok.2.1.6:

{-1} $(\text{addr}' + \text{delta}')\text{`offset} < \text{max\_linear\_offset}$

{-2} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-3} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$

{-4} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s')))))$
$=$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})(s'))))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.6`.

linear_resolve_same_page_address_ok.2.1.7:

{-1} (addr′ + delta′)‘offset < max_linear_offset

{-2} 0 ≤ (addr′ + delta′)‘offset

{-3} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))

{-4} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′)))))
2074 =
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))

Hiding formulas: (2 3),

Using lemma translate_memory_address,

Using lemma translate_memory_address2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_resolve_same_page_address_ok.2.1.7`.

`linear_resolve_same_page_address_ok.2.2:`

{-1} $0 \leq (\text{addr}' + \text{delta}')\text{`offset}$

{-2} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$

{-3} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(\text{addr}')) + \text{delta}' < \text{expt}(2,\ \text{min\_page}) \wedge$
$\text{OK?}(\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))))$
$\supset$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s')))))$
$=$
$\text{translate}(0,$
$\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})$
$(s'))))\text{`base\_addr},$
$\text{addr}' + \text{delta}',\ \text{ac}',$
$\text{segment\_to\_priv}(\text{data}(\text{read\_data}(\text{pm\_phy},\ \text{segment\_reg\_data\_type})(\text{CS})(s'))))$
$(\text{state}(\text{read\_data}(\text{pm\_phy},\ \text{pdbr\_data\_type})(\text{PDBR})$

Keeping (-4 -5 -6 -8 1) and hiding *,

Rewriting using delta_memory_address, matching in *,

This completes the proof of `linear_resolve_same_page_address_ok.2.2`.

`linear_resolve_same_page_address_ok.2.3:`

{-1} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))

{-2} rem(expt(2, min_page))(offset(addr′)) + delta′ < expt(2, min_page) ∧
OK?(translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))))
⊃
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′)))))
=
translate(0,
data(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)
(s′))))‘base_addr,
addr′ + delta′, ac′,
segment_to_priv(data(read_data(pm_phy, segment_reg_data_type)(CS)(s′))))
(state(read_data(pm_phy, pdbr_data_type)(PDBR)
(state(read_data(pm_phy, segment_reg_data_type)(CS)

Keeping (-3 -4 -6 1) and hiding *,
Expanding the definition of +,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_resolve_same_page_address_ok.2.3`.
Q.E.D.

C.117.67 Linear_Memory_Properties.linear_resolve_same_page_address_TCC1

Terse proof for `linear_resolve_same_page_address_TCC1`.
`linear_resolve_same_page_address_TCC1:`

{1} $\forall$ (s: Linear_memory[Physical_memory, pm_phy], addr: Memory_Address_4G, δ: nat,
ac: Memory_access):
rem(expt(2, min_page))(offset(addr)) + δ < expt(2, min_page) $\wedge$
OK?(linear_resolve(addr, ac)(s))
$\supset$
OK?[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](addr + δ, ac)(s))

Repeatedly Skolemizing and flattening,
Rewriting using linear_resolve_same_page_address_ok, matching in *,
This completes the proof of `linear_resolve_same_page_address_TCC1`.
Q.E.D.

C.117.68 Linear_Memory_Properties.linear_resolve_same_page_address

The LaTeX code for this proof is broken.

C.117.69 Linear_Memory_Properties.address_block_split_type

Terse proof for `address_block_split_type`.
`address_block_split_type:`

{1} $\forall$ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: list[Byte]):
is_linear_plain_memory?(pm) $\wedge$
(address_block(a, length(bl)) $\subseteq$ (pm`ro_addr $\cup$ pm`rw_addr))
$\supset$
every(λ (t: [Address, list[Byte]]):
Mem?(t`1`type_of) $\wedge$ 0 $\leq$ t`1`offset $\wedge$ t`1`offset < max_linear_offset)
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Using lemma split_type,
Using lemma address_block_in_memory,
Using lemma address_block_in_memory2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`address_block_split_type.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-3} 0 ≤ a′‘offset
{-4} a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
{-6} Mem?(a′‘type_of)
{-7} a′‘offset < max_linear_offset
{-8} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{1} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))

Hiding formulas: (-8),

Using lemma every_implied,

we get 2 subgoals:

`address_block_split_type.1.1:`

{-1} (∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a'))
⊃ Mem?(t_1‘1‘type_of) ∧ 0 ≤ t_1‘1‘offset ∧ t_1‘1‘offset < max_linear_offset)
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl$'$))
⊃
every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a', bl$'$))
{-2} is_linear_plain_memory?(pm$'$)
{-3} (address_block(a', length(bl$'$)) ⊆ (pm$'$‘ro_addr ∪ pm$'$‘rw_addr))
{-4} 0 ≤ a'‘offset
{-5} a' + length(bl$'$) ≤ reg_size(max_linear)(type_of(a'))
{-6} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl$'$))
{-7} Mem?(a'‘type_of)
{-8} a'‘offset < max_linear_offset

{1} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a', bl$'$))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Keeping (-1 -2 -3 1) and hiding *,

Installing automatic rewrites from: (max_linear! Mem!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `address_block_split_type.1.1`.

`address_block_split_type.1.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-3} 0 ≤ a′‘offset
{-4} a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
{-6} Mem?(a′‘type_of)
{-7} a′‘offset < max_linear_offset

{1} ∀ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ⊃ Mem?(min_linear‘type_of)
{2} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `address_block_split_type.1.2`.

`address_block_split_type.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-3} 0 ≤ a′‘offset
{-4} a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{-5} Mem?(a′‘type_of)
{-6} a′‘offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{1} reg_base(min_linear)(type_of(a′)) ≤ a′
{2} every(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ 0 ≤ t‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a′, bl′))

Keeping (-3 -5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `address_block_split_type.2`.

`address_block_split_type.3`:

{-1} is_linear_plain_memory?(pm′)
{-2} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-3} 0 ≤ a′`offset
{-4} Mem?(a′`type_of)
{-5} a′`offset < max_linear_offset
{-6} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{1} cons?(bl′)
{2} a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
{3} every(λ (t: [Address, list[Byte]]):
Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
(split(min_page, a′, bl′))

Using lemma split_null,
Keeping (-1 1 3) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `address_block_split_type.3`.
Q.E.D.

C.117.70 Linear_Memory_Properties.same_page_address_block_read_TCC1

Terse proof for `same_page_address_block_read_TCC1`.

`same_page_address_block_read_TCC1`:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
a_1, a_2: Memory_Address_4G, s_1, s_2: posnat):
is_linear_plain_memory?(pm) ∧
pm`states(s) ∧
union(pm`ro_addr, pm`rw_addr)(a_2) ∧
$a_1 \le a_2$ ∧
$a_2 + s_2 \le a_1 + s_1$ ∧
rem(expt(2, min_page))(offset(a_2)) + s_2 ≤ expt(2, min_page) ∧
(address_block(a_1, s_1) ⊆ (pm`ro_addr ∪ pm`rw_addr))
⊃ OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](a_2, Read)(s))

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_resolve_read_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `same_page_address_block_read_TCC1`.
Q.E.D.

C.117.71 Linear_Memory_Properties.same_page_address_block_read

Terse proof for `same_page_address_block_read`.

same_page_address_block_read:

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          a1, a2: Memory_Address_4G, s1, s2: posnat):
       is_linear_plain_memory?(pm) ∧
        pm‘states(s) ∧
         union(pm‘ro_addr, pm‘rw_addr)(a2) ∧
          a1 ≤ a2 ∧
           a2 + s2 ≤ a1 + s1 ∧
            rem(expt(2, min_page))(offset(a2)) + s2 ≤ expt(2, min_page) ∧
             (address_block(a1, s1) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
       ⊃
       (address_block(data(linear_resolve(a2, Read)(s)), s2) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
```

Installing automatic rewrites from: singleton member subset?
Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-4 -6 -7 -8 -9),
Instantiating the top quantifier in -3 with the terms: (a2!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

same_page_address_block_read.1:

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states(s′)
{-3}  OK?(linear_resolve(a2′, Read)(s′))
{-4}  OK?(linear_resolve(a2′, Execute)(s′))
{-5}  ∀ (x: Memory_Address_4G):
        virt_to_phys_range(s′,
                           (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
                           (singleton(Read) ∪ singleton(Execute)))
                          (x)
        ⇒
        restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-6}  Mem?(a1′‘type_of)
{-7}  0 ≤ a1′‘offset
{-8}  a1′‘offset < max_linear_offset
{-9}  Mem?(a2′‘type_of)
{-10} 0 ≤ a2′‘offset
{-11} a2′‘offset < max_linear_offset
{-12} s1′ > 0
{-13} s2′ > 0
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(a2′)
{-15} a1′ ≤ a2′
{-16} a2′ + s2′ ≤ a1′ + s1′
{-17} rem(expt(2, min_page))(offset(a2′)) + s2′ ≤ expt(2, min_page)
{-18} ∀ (x: Address): address_block(a1′, s1′)(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
-------------------------------------------------------------------------------------
 {1}  ∀ (x: Address):
        address_block(data(linear_resolve(a2′, Read)(s′)), s2′)(x) ⇒
         union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x)
```

Repeatedly Skolemizing and flattening,

Case splitting on offset(x!1) - offset (data[Physical_memory, Address] (linear_resolve[Physical_memory, pm_phy](a2!1, Read)(s!1))) >= 0,

we get 2 subgoals:

`same_page_address_block_read.1.1:`

{-1} $\text{offset}(x') - \text{offset}(\text{data}\ [\text{Physical\_memory},\ \text{Address}](\text{linear\_resolve}\ [\text{Physical\_memory},\ \text{pm\_phy}](a_2',\ \text{Read})($
≥ 0
{-2} $\text{is\_linear\_plain\_memory?}(\text{pm}')$
{-3} $\text{pm}'\text{`states}(s')$
{-4} $\text{OK?}(\text{linear\_resolve}(a_2',\ \text{Read})(s'))$
{-5} $\text{OK?}(\text{linear\_resolve}(a_2',\ \text{Execute})(s'))$
{-6} $\forall\ (x\colon\ \text{Memory\_Address\_4G})\colon$
$\text{virt\_to\_phys\_range}(s',$
$(\text{restrict}[\text{Address},\ \text{Memory\_Address\_4G},\ \text{boolean}](\text{pm}'\text{`ro\_addr}) \cup \text{restrict}\ [\text{Addre}$
$(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$
(x)
$\Rightarrow$
$\text{restrict}[\text{Address},\ \text{Memory\_Address\_4G},\ \text{boolean}]((\text{pm\_phy`ro\_addr} \cup \text{pm\_phy`rw\_addr}))(x)$
{-7} $\text{Mem?}(a_1'\text{`type\_of})$
{-8} $0 \leq a_1'\text{`offset}$
{-9} $a_1'\text{`offset} < \text{max\_linear\_offset}$
{-10} $\text{Mem?}(a_2'\text{`type\_of})$
{-11} $0 \leq a_2'\text{`offset}$
{-12} $a_2'\text{`offset} < \text{max\_linear\_offset}$
{-13} $s_1' > 0$
{-14} $s_2' > 0$
{-15} $\text{union}(\text{pm}'\text{`ro\_addr},\ \text{pm}'\text{`rw\_addr})(a_2')$
{-16} $a_1' \leq a_2'$
{-17} $a_2' + s_2' \leq a_1' + s_1'$
{-18} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2,\ \text{min\_page})$
{-19} $\forall\ (x\colon\ \text{Address})\colon\ \text{address\_block}(a_1',\ s_1')(x) \Rightarrow \text{union}(\text{pm}'\text{`ro\_addr},\ \text{pm}'\text{`rw\_addr})(x)$
{-20} $\text{address\_block}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s')),\ s_2')(x')$

{1} $\text{union}(\text{pm\_phy`ro\_addr},\ \text{pm\_phy`rw\_addr})(x')$

Using lemma linear_resolve_same_page_address,

Case splitting on rem(expt(2, min_page))(offset(a2!1)) + offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1))) < expt(2, min_page),

we get 2 subgoals:

`same_page_address_block_read.1.1.1:`

{-1} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Read)(s')))
< expt(2, min_page)
{-2} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Read)(s')))
< expt(2, min_page)
∧ OK?(linear_resolve(a_2', Read)(s'))
⊃
data(linear_resolve(a_2', Read)(s')) + offset(x') − offset(data(linear_resolve(a_2', Read)(s')))
=
data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))),
Read)
(s'))
{-3} offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physical_memory, pm_phy](
≥ 0
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states(s')
{-6} OK?(linear_resolve(a_2', Read)(s'))
{-7} OK?(linear_resolve(a_2', Execute)(s'))
{-8} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s',
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
(singleton(Read) ∪ singleton(Execute)))
(x)
⇒
restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-9} Mem?(a_1'‘type_of)
{-10} 0 ≤ a_1'‘offset
{-11} a_1'‘offset < max_linear_offset
{-12} Mem?(a_2'‘type_of)
{-13} 0 ≤ a_2'‘offset
{-14} a_2'‘offset < max_linear_offset
{-15} s_1' > 0
{-16} s_2' > 0
{-17} union(pm′‘ro_addr, pm′‘rw_addr)(a_2')
{-18} a_1' ≤ a_2'
{-19} $a_2' + s_2'$ ≤ $a_1' + s_1'$
{-20} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-21} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
{-22} address_block(data(linear_resolve(a_2', Read)(s')), s_2')(x')

{1} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on data(linear_resolve(a2!1, Read)(s!1)) + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1)))) = x!1,

we get 2 subgoals:

same_page_address_block_read.1.1.1.1:

{-1} data(linear_resolve(a'_2, Read)(s')) + (offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))))
= x'
{-2} rem(expt(2, min_page))(offset(a'_2)) + offset(x') − offset(data(linear_resolve(a'_2, Read)(s')))
< expt(2, min_page)
{-3} OK?(linear_resolve(a'_2, Read)(s'))
{-4} data(linear_resolve(a'_2, Read)(s')) + offset(x') − offset(data(linear_resolve(a'_2, Read)(s')))
=
data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))),
Read)
(s'))
{-5} offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physical_memory, pm_phy](a'_2, Read)(
≥ 0
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states(s')
{-8} OK?(linear_resolve(a'_2, Execute)(s'))
{-9} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s',
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Addre
(singleton(Read) ∪ singleton(Execute)))
(x)
⇒
restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-10} Mem?(a'_1‘type_of)
{-11} 0 ≤ a'_1‘offset
{-12} a'_1‘offset < max_linear_offset
{-13} Mem?(a'_2‘type_of)
{-14} 0 ≤ a'_2‘offset
{-15} a'_2‘offset < max_linear_offset
{-16} s'_1 > 0
{-17} s'_2 > 0
{-18} union(pm′‘ro_addr, pm′‘rw_addr)(a'_2)
{-19} a'_1 ≤ a'_2
{-20} $a'_2 + s'_2$ ≤ $a'_1 + s'_1$
{-21} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-22} ∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
{-23} address_block(data(linear_resolve(a'_2, Read)(s')), s'_2)(x')

{1} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x')

Replacing using formula -1,

Replacing using formula -4,

Case splitting on OK?(linear_resolve(a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1)))), Read) (s!1)),

we get 2 subgoals:

same_page_address_block_read.1.1.1.1.1:

{-1} OK?(linear_resolve(a'_2 + (offset(x') − offset(data(linear_resolve(a'_2, Read)(s')))), Read)(s'))
{-2} data(linear_resolve(a'_2, Read)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_
= data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s'))
{-3} rem(expt(2, min_page))(offset(a'_2)) + offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear
< expt(2, min_page)
{-4} OK?(linear_resolve(a'_2, Read)(s'))
{-5} x' = data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s'))
{-6} offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))
≥ 0
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s')
{-9} OK?(linear_resolve(a'_2, Execute)(s'))
{-10} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s', (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
(singleton(Read) ∪ singleton(Execute)))(x)
⇒
restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-11} Mem?(a'_1‘type_of)
{-12} 0 ≤ a'_1‘offset
{-13} a'_1‘offset < max_linear_offset
{-14} Mem?(a'_2‘type_of)
{-15} 0 ≤ a'_2‘offset
{-16} a'_2‘offset < max_linear_offset
{-17} s'_1 > 0
{-18} s'_2 > 0
{-19} union(pm′‘ro_addr, pm′‘rw_addr)(a'_2)
{-20} a'_1 ≤ a'_2
{-21} $a'_2 + s'_2 \le a'_1 + s'_1$
{-22} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-23} ∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
{-24} address_block(data(linear_resolve(a'_2, Read)(s')), s'_2)(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))

{1} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))

Instantiating the top quantifier in -10 with the terms: (data(linear_resolve(a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1))))), Read) (s!1))),
we get 2 subgoals:

same_page_address_block_read.1.1.1.1.1.1:

{-1} OK?(linear_resolve($a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)($s'$))

{-2} $\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
$=$
$\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))$

{-3} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(a_2')) + \text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
$< \text{expt}(2, \text{min\_page})$

{-4} OK?(linear_resolve(a_2', Read)(s'))

{-5} $x' =$
$\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))$

{-6} $\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))) - \text{offset}(\text{dat}$
≥ 0

{-7} is_linear_plain_memory?(pm′)

{-8} pm′`states(s')

{-9} OK?(linear_resolve(a_2', Execute)(s'))

{-10} virt_to_phys_range(s',
(restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Address,
(singleton(Read) ∪ singleton(Execute)))
(data(linear_resolve($a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)($s'$)))
⇒
restrict[Address, Memory_Address_4G, boolean]
((pm_phy`ro_addr ∪ pm_phy`rw_addr))
(data(linear_resolve($a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)($s'$)))

{-11} Mem?(a_1'`type_of)

{-12} $0 \leq a_1'$`offset

{-13} a_1'`offset $<$ max_linear_offset

{-14} Mem?(a_2'`type_of)

{-15} $0 \leq a_2'$`offset

{-16} a_2'`offset $<$ max_linear_offset

{-17} $s_1' > 0$

{-18} $s_2' > 0$

{-19} union(pm′`ro_addr, pm′`rw_addr)(a_2')

{-20} $a_1' \leq a_2'$

{-21} $a_2' + s_2' \leq a_1' + s_1'$

{-22} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2, \text{min\_page})$

{-23} $\forall$ (x: Address): address_block(a_1', s_1')(x) ⇒ union(pm′`ro_addr, pm′`rw_addr)(x)

{-24} address_block(data(linear_resolve(a_2', Read)(s')), s_2')
(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s')$))

{1} union(pm_phy`ro_addr, pm_phy`rw_addr)
(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s')$))

Expanding the definition of restrict,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of virt_to_phys_range,

Instantiating the top quantifier in 1 with the terms: (a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1)))) Read),

we get 3 subgoals:

`same_page_address_block_read.1.1.1.1.1.1.1:`

{-1} $\mathrm{OK?}(\mathrm{linear\_resolve}(a'_2 + (\mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')))),\ \mathrm{Read})(s'))$

{-2} $\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')) + (\mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_}$
$=$
$\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')))),\ \mathrm{Read})(s'))$

{-3} $\mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')))),\ \mathrm{Read})(s')))$
$< \mathrm{expt}(2,\ \mathrm{min\_page})$

{-4} $\mathrm{OK?}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s'))$

{-5} $x' =$
$\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')))),\ \mathrm{Read})(s'))$

{-6} $\mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')))),\ \mathrm{Read})(s')))$
≥ 0

{-7} $\mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}')$

{-8} $\mathrm{pm}'\text{`states}(s')$

{-9} $\mathrm{OK?}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Execute})(s'))$

{-10} $\mathrm{Mem?}(a'_1\text{`type\_of})$

{-11} $0 \leq a'_1\text{`offset}$

{-12} $a'_1\text{`offset} < \mathrm{max\_linear\_offset}$

{-13} $\mathrm{Mem?}(a'_2\text{`type\_of})$

{-14} $0 \leq a'_2\text{`offset}$

{-15} $a'_2\text{`offset} < \mathrm{max\_linear\_offset}$

{-16} $s'_1 > 0$

{-17} $s'_2 > 0$

{-18} $\mathrm{union}(\mathrm{pm}'\text{`ro\_addr},\ \mathrm{pm}'\text{`rw\_addr})(a'_2)$

{-19} $a'_1 \leq a'_2$

{-20} $a'_2 + s'_2 \leq a'_1 + s'_1$

{-21} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{min\_page}))(\mathrm{offset}(a'_2)) + s'_2 \leq \mathrm{expt}(2,\ \mathrm{min\_page})$

{-22} $\forall\ (x\colon \mathrm{Address})\colon \mathrm{address\_block}(a'_1,\ s'_1)(x) \Rightarrow \mathrm{union}(\mathrm{pm}'\text{`ro\_addr},\ \mathrm{pm}'\text{`rw\_addr})(x)$

{-23} $\mathrm{address\_block}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')),\ s'_2)$
$(\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s'))),$
$\mathrm{Read})(s')))$

{1} $\mathrm{OK?}(\mathrm{linear\_resolve}(a'_2 + (\mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s')))),\ \mathrm{Read})(s'))$

{2} $\mathrm{union}(\mathrm{pm\_phy}\text{`ro\_addr},\ \mathrm{pm\_phy}\text{`rw\_addr})$
$(\mathrm{data}(\mathrm{linear\_resolve}(a'_2 + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a'_2,\ \mathrm{Read})(s'))),$
$\mathrm{Read})(s')))$

which is trivially true.

This completes the proof of `same_page_address_block_read.1.1.1.1.1.1.1`.

`same_page_address_block_read.1.1.1.1.1.1.2`:

{-1} OK?(linear_resolve($a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)$(s')$)
{-2} $\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s')) + (\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
 $=$
 $\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))$
{-3} $\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))) + \text{rem}(\text{expt}$
 $< \text{expt}(2, \text{min\_page})$
{-4} OK?(linear_resolve(a_2', Read)(s'))
{-5} $x' =$
 $\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))$
{-6} $\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))) - \text{offset}(\text{dat}$
 ≥ 0
{-7} is_linear_plain_memory?(pm′)
{-8} pm′`states(s')
{-9} OK?(linear_resolve(a_2', Execute)(s'))
{-10} Mem?(a_1'`type_of)
{-11} $0 \leq a_1'$`offset
{-12} a_1'`offset $<$ max_linear_offset
{-13} Mem?(a_2'`type_of)
{-14} $0 \leq a_2'$`offset
{-15} a_2'`offset $<$ max_linear_offset
{-16} $s_1' > 0$
{-17} $s_2' > 0$
{-18} union(pm′`ro_addr, pm′`rw_addr)(a_2')
{-19} $a_1' \leq a_2'$
{-20} $a_2' + s_2' \leq a_1' + s_1'$
{-21} $\text{rem}(\text{expt}(2, \text{min\_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2, \text{min\_page})$
{-22} $\forall$ (x: Address): address_block(a_1', s_1')(x) $\Rightarrow$ union(pm′`ro_addr, pm′`rw_addr)(x)
{-23} address_block(data(linear_resolve(a_2', Read)(s')), s_2')
 (data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))),$ Read)(s')))

{1} union[Memory_access]
 (singleton[Memory_access](Read), singleton[Memory_access](Execute))(Read)
{2} union(pm_phy`ro_addr, pm_phy`rw_addr)
 (data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))),$ Read)(s')))

Using lemma linear_resolve_same_page_address_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of union,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `same_page_address_block_read.1.1.1.1.1.1.2`.

`same_page_address_block_read.1.1.1.1.1.1.3:`

{-1} OK?(linear_resolve(a'_2 + (offset(x') − offset(data(linear_resolve(a'_2, Read)(s')))), Read)(s'))
{-2} data(linear_resolve(a'_2, Read)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_
= data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s'))
{-3} offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))
< expt(2, min_page)
{-4} OK?(linear_resolve(a'_2, Read)(s'))
{-5} x' = data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s'))
{-6} offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))
≥ 0
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s')
{-9} OK?(linear_resolve(a'_2, Execute)(s'))
{-10} Mem?(a'_1‘type_of)
{-11} 0 ≤ a'_1‘offset
{-12} a'_1‘offset < max_linear_offset
{-13} Mem?(a'_2‘type_of)
{-14} 0 ≤ a'_2‘offset
{-15} a'_2‘offset < max_linear_offset
{-16} s'_1 > 0
{-17} s'_2 > 0
{-18} union(pm′‘ro_addr, pm′‘rw_addr)(a'_2)
{-19} a'_1 ≤ a'_2
{-20} $a'_2 + s'_2$ ≤ $a'_1 + s'_1$
{-21} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-22} ∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
{-23} address_block(data(linear_resolve(a'_2, Read)(s')), s'_2)
(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))

{1} union[Memory_Address_4G]
(λ (s: Memory_Address_4G): pm′‘ro_addr(s),
λ (s: Memory_Address_4G): pm′‘rw_addr(s))
(a'_2 + (offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physical_memory
{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)
(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Read)(s'))), Read)(s')))

Instantiating the top quantifier in -22 with the terms: (a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Read)(s!1))))),

Hiding formulas: (-3 -4 -6 -9 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `same_page_address_block_read.1.1.1.1.1.1.3`.

same_page_address_block_read.1.1.1.1.1.2:

{-1} OK?(linear_resolve($a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)(s'))

{-2} data(linear_resolve(a_2', Read)(s')) + (offset(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
=
data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)(s'))

{-3} rem(expt(2, min_page))(offset(a_2')) + offset(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
$<$ expt(2, min_page)

{-4} OK?(linear_resolve(a_2', Read)(s'))

{-5} $x' =$
data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))))$, Read)(s'))

{-6} offset(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a_2', \text{Read})(s'))), \text{Read})(s'))) - offset(dat
≥ 0

{-7} is_linear_plain_memory?(pm')

{-8} pm'`states(s')

{-9} OK?(linear_resolve(a_2', Execute)(s'))

{-10} Mem?(a_1'`type_of)

{-11} $0 \leq a_1'$`offset

{-12} a_1'`offset $<$ max_linear_offset

{-13} Mem?(a_2'`type_of)

{-14} $0 \leq a_2'$`offset

{-15} a_2'`offset $<$ max_linear_offset

{-16} $s_1' > 0$

{-17} $s_2' > 0$

{-18} union(pm'`ro_addr, pm'`rw_addr)(a_2')

{-19} $a_1' \leq a_2'$

{-20} $a_2' + s_2' \leq a_1' + s_1'$

{-21} rem(expt(2, min_page))(offset(a_2')) + $s_2' \leq$ expt(2, min_page)

{-22} $\forall$ (x: Address): address_block(a_1', s_1')(x) $\Rightarrow$ union(pm'`ro_addr, pm'`rw_addr)(x)

{-23} address_block(data(linear_resolve(a_2', Read)(s')), s_2')
(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s')))$

{1} Mem?(data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy]
($a_2' + (\text{offset}(x') - \text{offset}(\text{data}$ [Physical_memory, Address](linear_resolve [Physical_memo
Read)
(s'))`type_of)
$\wedge$
$0 \leq$
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy]
($a_2' + (\text{offset}(x') - \text{offset}(\text{data}$ [Physical_memory, Address](linear_resolve [Physical_memory,
Read)
(s'))`offset
$\wedge$
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy]
($a_2' + (\text{offset}(x') - \text{offset}(\text{data}$ [Physical_memory, Address](linear_resolve [Physical_memory,
Read)
(s'))`offset
$<$ max_linear_offset

{2} union(pm_phy`ro_addr, pm_phy`rw_addr)
(data(linear_resolve($a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{Read})(s'))), \text{Read})(s')))$

Using lemma linear_resolve_memory_address,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `same_page_address_block_read.1.1.1.1.1.2`.
`same_page_address_block_read.1.1.1.1.2:`

{-1} data(linear_resolve(a_2', Read)(s')) + (offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear
=
data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))),
Read)
(s'))
{-2} rem(expt(2, min_page))(offset(a_2')) + offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear
< expt(2, min_page)
{-3} OK?(linear_resolve(a_2', Read)(s'))
{-4} x' =
data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))),
Read)
(s'))
{-5} offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))), Read)(s')))
≥ 0
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states(s')
{-8} OK?(linear_resolve(a_2', Execute)(s'))
{-9} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s',
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
(singleton(Read) ∪ singleton(Execute)))
(x)
⇒
restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-10} Mem?(a_1'‘type_of)
{-11} 0 ≤ a_1'‘offset
{-12} a_1'‘offset < max_linear_offset
{-13} Mem?(a_2'‘type_of)
{-14} 0 ≤ a_2'‘offset
{-15} a_2'‘offset < max_linear_offset
{-16} s_1' > 0
{-17} s_2' > 0
{-18} union(pm′‘ro_addr, pm′‘rw_addr)(a_2')
{-19} a_1' ≤ a_2'
{-20} $a_2' + s_2'$ ≤ $a_1' + s_1'$
{-21} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-22} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
{-23} address_block(data(linear_resolve(a_2', Read)(s')), s_2')
(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))),
Read)
(s')))

{1} OK?(linear_resolve(a_2' + (offset(x') − offset(data(linear_resolve(a_2', Read)(s')))),
Read)
(s'))
{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)
(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))),
Read)
(s')))

Using lemma linear_resolve_same_page_address_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `same_page_address_block_read.1.1.1.1.2`.

`same_page_address_block_read.1.1.1.2:`

{-1} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(a_2')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s')))$
$< \text{expt}(2, \text{min\_page})$

{-2} $\text{OK?}(\text{linear\_resolve}(a_2', \text{Read})(s'))$

{-3} $\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s')) + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s')))$
$=$
$\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s'))),$
$\text{Read})$
$(s'))$

{-4} $\text{offset}(x') - \text{offset}(\text{data}\ [\text{Physical\_memory},\ \text{Address}](\text{linear\_resolve}\ [\text{Physical\_memory},\ \text{pm\_phy}](a_2',\ \text{Read})($
≥ 0

{-5} $\text{is\_linear\_plain\_memory?}(\text{pm}')$

{-6} $\text{pm}'\text{`states}(s')$

{-7} $\text{OK?}(\text{linear\_resolve}(a_2', \text{Execute})(s'))$

{-8} $\forall\ (x\text{: Memory\_Address\_4G}):$
$\text{virt\_to\_phys\_range}(s',$
$(\text{restrict}[\text{Address},\ \text{Memory\_Address\_4G},\ \text{boolean}](\text{pm}'\text{`ro\_addr}) \cup \text{restrict}\ [\text{Addre}$
$(\text{singleton}(\text{Read}) \cup \text{singleton}(\text{Execute})))$
(x)
$\Rightarrow$
$\text{restrict}[\text{Address}, \text{Memory\_Address\_4G}, \text{boolean}]((\text{pm\_phy}\text{`ro\_addr} \cup \text{pm\_phy}\text{`rw\_addr}))(x)$

{-9} $\text{Mem?}(a_1'\text{`type\_of})$

{-10} $0 \leq a_1'\text{`offset}$

{-11} $a_1'\text{`offset} < \text{max\_linear\_offset}$

{-12} $\text{Mem?}(a_2'\text{`type\_of})$

{-13} $0 \leq a_2'\text{`offset}$

{-14} $a_2'\text{`offset} < \text{max\_linear\_offset}$

{-15} $s_1' > 0$

{-16} $s_2' > 0$

{-17} $\text{union}(\text{pm}'\text{`ro\_addr}, \text{pm}'\text{`rw\_addr})(a_2')$

{-18} $a_1' \leq a_2'$

{-19} $a_2' + s_2' \leq a_1' + s_1'$

{-20} $\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2, \text{min\_page})$

{-21} $\forall\ (x\text{: Address}):\ \text{address\_block}(a_1',\ s_1')(x) \Rightarrow \text{union}(\text{pm}'\text{`ro\_addr}, \text{pm}'\text{`rw\_addr})(x)$

{-22} $\text{address\_block}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s')),\ s_2')(x')$

{1} $\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s')) + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Read})(s'))))$
$= x'$

{2} $\text{union}(\text{pm\_phy}\text{`ro\_addr}, \text{pm\_phy}\text{`rw\_addr})(x')$

Expanding the definition of +,

Applying decompose-equality,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `same_page_address_block_read.1.1.1.2`.

same_page_address_block_read.1.1.2:

{-1} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Read)(s')))
< expt(2, min_page)
∧ OK?(linear_resolve(a_2', Read)(s'))
⊃
data(linear_resolve(a_2', Read)(s')) + offset(x') − offset(data(linear_resolve(a_2', Read)(s')))
=
data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Read)(s'))),
Read)
(s'))

{-2} offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physical_memory, pm_phy](
≥ 0

{-3} is_linear_plain_memory?(pm′)

{-4} pm′‘states(s')

{-5} OK?(linear_resolve(a_2', Read)(s'))

{-6} OK?(linear_resolve(a_2', Execute)(s'))

{-7} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s',
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
(singleton(Read) ∪ singleton(Execute)))
(x)
⇒
restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)

{-8} Mem?(a_1'‘type_of)

{-9} 0 ≤ a_1'‘offset

{-10} a_1'‘offset < max_linear_offset

{-11} Mem?(a_2'‘type_of)

{-12} 0 ≤ a_2'‘offset

{-13} a_2'‘offset < max_linear_offset

{-14} s_1' > 0

{-15} s_2' > 0

{-16} union(pm′‘ro_addr, pm′‘rw_addr)(a_2')

{-17} a_1' ≤ a_2'

{-18} $a_2' + s_2'$ ≤ $a_1' + s_1'$

{-19} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)

{-20} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)

{-21} address_block(data(linear_resolve(a_2', Read)(s')), s_2')(x')

{1} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Read)(s')))
< expt(2, min_page)

{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x')

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of same_page_address_block_read.1.1.2.

`same_page_address_block_read.1.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} OK?(linear_resolve(a′2, Read)(s′))
{-4} OK?(linear_resolve(a′2, Execute)(s′))
{-5} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s′,
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Addre
(singleton(Read) ∪ singleton(Execute)))
(x)
⇒
restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-6} Mem?(a′1‘type_of)
{-7} 0 ≤ a′1‘offset
{-8} a′1‘offset < max_linear_offset
{-9} Mem?(a′2‘type_of)
{-10} 0 ≤ a′2‘offset
{-11} a′2‘offset < max_linear_offset
{-12} s′1 > 0
{-13} s′2 > 0
{-14} union(pm′‘ro_addr, pm′‘rw_addr)(a′2)
{-15} a′1 ≤ a′2
{-16} a′2 + s′2 ≤ a′1 + s′1
{-17} rem(expt(2, min_page))(offset(a′2)) + s′2 ≤ expt(2, min_page)
{-18} ∀ (x: Address): address_block(a′1, s′1)(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
{-19} address_block(data(linear_resolve(a′2, Read)(s′)), s′2)(x′)

{1} offset(x′) − offset(data [Physical_memory, Address](linear_resolve [Physical_memory, pm_phy](a′2, Read)(
≥ 0
{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x′)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `same_page_address_block_read.1.2`.

`same_page_address_block_read.2:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  pm′‘states(s′)
{-3}  ∀ (x: Memory_Address_4G):
        virt_to_phys_range(s′,
                           (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restr
                           (singleton(Read) ∪ singleton(Execute)))
                          (x)
        ⇒
        restrict[Address, Memory_Address_4G, boolean]((pm_phy‘ro_addr ∪ pm_phy‘rw_addr))(x)
{-4}  Mem?(a′1‘type_of)
{-5}  0 ≤ a′1‘offset
{-6}  a′1‘offset < max_linear_offset
{-7}  Mem?(a′2‘type_of)
{-8}  0 ≤ a′2‘offset
{-9}  a′2‘offset < max_linear_offset
{-10} s′1 > 0
{-11} s′2 > 0
{-12} union(pm′‘ro_addr, pm′‘rw_addr)(a′2)
{-13} a′1 ≤ a′2
{-14} a′2 + s′2 ≤ a′1 + s′1
{-15} rem(expt(2, min_page))(offset(a′2)) + s′2 ≤ expt(2, min_page)
{-16} ∀ (x: Address): address_block(a′1, s′1)(x) ⇒ union(pm′‘ro_addr, pm′‘rw_addr)(x)
------------------------------------------------------------------------------
 {1}  union(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr),
            restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr))
           (a′2)
 {2}  ∀ (x: Address):
        address_block(data(linear_resolve(a′2, Read)(s′)), s′2)(x) ⇒
        union(pm_phy‘ro_addr, pm_phy‘rw_addr)(x)
```

Keeping (-12 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `same_page_address_block_read.2`.

Q.E.D.

C.117.72 Linear_Memory_Properties.same_page_address_block_write_TCC1

Terse proof for `same_page_address_block_write_TCC1`.

`same_page_address_block_write_TCC1`:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
a_1, a_2: Memory_Address_4G, s_1, s_2: posnat):
is_linear_plain_memory?(pm) ∧
pm‘states(s) ∧
pm‘rw_addr(a_2) ∧
$a_1 \leq a_2$ ∧
$a_2 + s_2 \leq a_1 + s_1$ ∧
rem(expt(2, min_page))(offset(a_2)) + s_2 ≤ expt(2, min_page) ∧
(address_block(a_1, s_1) ⊆ pm‘rw_addr)
⊃
OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](a_2, Write)(s))

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_resolve_write_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `same_page_address_block_write_TCC1`.

Q.E.D.

C.117.73 Linear_Memory_Properties.same_page_address_block_write

Terse proof for `same_page_address_block_write`.

`same_page_address_block_write`:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
a_1, a_2: Memory_Address_4G, s_1, s_2: posnat):
is_linear_plain_memory?(pm) ∧
pm‘states(s) ∧
pm‘rw_addr(a_2) ∧
$a_1 \leq a_2$ ∧
$a_2 + s_2 \leq a_1 + s_1$ ∧
rem(expt(2, min_page))(offset(a_2)) + s_2 ≤ expt(2, min_page) ∧
(address_block(a_1, s_1) ⊆ pm‘rw_addr)
⊃ (address_block(data(linear_resolve(a_2, Write)(s)), s_2) ⊆ pm_phy‘rw_addr)

Installing automatic rewrites from: singleton member subset?

Repeatedly Skolemizing and flattening,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-3 -5 -7 -8 -9),

Instantiating the top quantifier in -3 with the terms: (a2!1),

Repeatedly Skolemizing and flattening,

Case splitting on offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1))) >= 0,

we get 2 subgoals:

same_page_address_block_write.1:

{-1}	$\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',\ \text{Write})(s'))) \geq 0$
{-2}	$\text{is\_linear\_plain\_memory?}(\text{pm}')$
{-3}	$\text{pm}'\text{`states}(s')$
{-4}	$\text{restrict}[\text{Address}, \text{Memory\_Address\_4G}, \text{boolean}](\text{pm}'\text{`rw\_addr})(a_2') \supset$ $\text{OK?}(\text{linear\_resolve}(a_2', \text{Write})(s'))$
{-5}	$\forall\ (x\colon \text{Memory\_Address\_4G})\colon$ $\text{virt\_to\_phys\_range}(s', \text{restrict}[\text{Address}, \text{Memory\_Address\_4G}, \text{boolean}](\text{pm}'\text{`rw\_addr}), \text{singleton(Write)})(x)$ $\Rightarrow \text{restrict}[\text{Address}, \text{Memory\_Address\_4G}, \text{boolean}](\text{pm\_phy}\text{`rw\_addr})(x)$
{-6}	$\text{Mem?}(a_1'\text{`type\_of})$
{-7}	$0 \leq a_1'\text{`offset}$
{-8}	$a_1'\text{`offset} < \text{max\_linear\_offset}$
{-9}	$\text{Mem?}(a_2'\text{`type\_of})$
{-10}	$0 \leq a_2'\text{`offset}$
{-11}	$a_2'\text{`offset} < \text{max\_linear\_offset}$
{-12}	$s_1' > 0$
{-13}	$s_2' > 0$
{-14}	$\text{pm}'\text{`rw\_addr}(a_2')$
{-15}	$a_1' \leq a_2'$
{-16}	$a_2' + s_2' \leq a_1' + s_1'$
{-17}	$\text{rem}(\text{expt}(2,\ \text{min\_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2, \text{min\_page})$
{-18}	$\forall\ (x\colon \text{Address})\colon \text{address\_block}(a_1', s_1')(x) \Rightarrow \text{pm}'\text{`rw\_addr}(x)$
{-19}	$\text{address\_block}(\text{data}(\text{linear\_resolve}(a_2', \text{Write})(s')), s_2')(x')$
{1}	$\text{pm\_phy}\text{`rw\_addr}(x')$

Using lemma linear_resolve_same_page_address,

Case splitting on rem(expt(2, min_page))(offset(a2!1)) + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1)))) < expt(2, min_page),

we get 2 subgoals:

same_page_address_block_write.1.1:

{-1} rem(expt(2, min_page))(offset(a_2')) + (offset(x') − offset(data(linear_resolve(a_2', Write)(s')))) < expt(2, min_page)

{-2} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) < expt(2, min_page) ∧ OK?(linear_resolve(a_2', Write)(s')) ⊃ data(linear_resolve(a_2', Write)(s')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) = data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))

{-3} offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) ≥ 0

{-4} is_linear_plain_memory?(pm′)

{-5} pm′`states(s')

{-6} restrict[Address, Memory_Address_4G, boolean](pm′`rw_addr)(a_2') ⊃ OK?(linear_resolve(a_2', Write)(s'))

{-7} ∀ (x: Memory_Address_4G): virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′`rw_addr), singleton(Write))($x$) ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy`rw_addr)(x)

{-8} Mem?(a_1'`type_of)

{-9} 0 ≤ a_1'`offset

{-10} a_1'`offset < max_linear_offset

{-11} Mem?(a_2'`type_of)

{-12} 0 ≤ a_2'`offset

{-13} a_2'`offset < max_linear_offset

{-14} s_1' > 0

{-15} s_2' > 0

{-16} pm′`rw_addr(a_2')

{-17} a_1' ≤ a_2'

{-18} a_2' + s_2' ≤ a_1' + s_1'

{-19} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)

{-20} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′`rw_addr(x)

{-21} address_block(data(linear_resolve(a_2', Write)(s')), s_2')(x')

{1} pm_phy`rw_addr(x')

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

same_page_address_block_write.1.1.1:

{-1}	rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) < expt(2, min_page)
{-2}	OK?(linear_resolve(a_2', Write)(s'))
{-3}	data(linear_resolve(a_2', Write)(s')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) = data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-4}	offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) ≥ 0
{-5}	is_linear_plain_memory?(pm′)
{-6}	pm′‘states(s')
{-7}	∀ (x: Memory_Address_4G): virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x) ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-8}	Mem?(a_1'‘type_of)
{-9}	0 ≤ a_1'‘offset
{-10}	a_1'‘offset < max_linear_offset
{-11}	Mem?(a_2'‘type_of)
{-12}	0 ≤ a_2'‘offset
{-13}	a_2'‘offset < max_linear_offset
{-14}	s_1' > 0
{-15}	s_2' > 0
{-16}	pm′‘rw_addr(a_2')
{-17}	a_1' ≤ a_2'
{-18}	a_2' + s_2' ≤ a_1' + s_1'
{-19}	rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-20}	∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-21}	address_block(data(linear_resolve(a_2', Write)(s')), s_2')(x')
{1}	pm_phy‘rw_addr(x')

Case splitting on data(linear_resolve(a2!1, Write)(s!1)) + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1)))) = x!1,

we get 2 subgoals:

same_page_address_block_write.1.1.1.1:

{-1}	data(linear_resolve(a'_2, Write)(s')) + (offset(x') − offset(data(linear_resolve(a'_2, Write)(s')))) = x'
{-2}	rem(expt(2, min_page))(offset(a'_2)) + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))) < expt(2, min_page)
{-3}	OK?(linear_resolve(a'_2, Write)(s'))
{-4}	data(linear_resolve(a'_2, Write)(s')) + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))) = data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s'))
{-5}	offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))) ≥ 0
{-6}	is_linear_plain_memory?(pm′)
{-7}	pm′‘states(s')
{-8}	∀ (x: Memory_Address_4G): virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x) ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-9}	Mem?(a'_1‘type_of)
{-10}	0 ≤ a'_1‘offset
{-11}	a'_1‘offset < max_linear_offset
{-12}	Mem?(a'_2‘type_of)
{-13}	0 ≤ a'_2‘offset
{-14}	a'_2‘offset < max_linear_offset
{-15}	s'_1 > 0
{-16}	s'_2 > 0
{-17}	pm′‘rw_addr(a'_2)
{-18}	a'_1 ≤ a'_2
{-19}	a'_2 + s'_2 ≤ a'_1 + s'_1
{-20}	rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-21}	∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ pm′‘rw_addr(x)
{-22}	address_block(data(linear_resolve(a'_2, Write)(s')), s'_2)(x')
{1}	pm_phy‘rw_addr(x')

Replacing using formula -1,

Replacing using formula -4,

Case splitting on OK?(linear_resolve(a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1)))), Write) (s!1)),

we get 2 subgoals:

same_page_address_block_write.1.1.1.1.1:

{-1} OK?(linear_resolve(a'_2 + (offset(x') − offset(data(linear_resolve(a'_2, Write)(s')))), Write)(s'))
{-2} data(linear_resolve(a'_2, Write)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear
= data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s'))
{-3} rem(expt(2, min_page))(offset(a'_2)) + offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear
< expt(2, min_page)
{-4} OK?(linear_resolve(a'_2, Write)(s'))
{-5} x' = data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s'))
{-6} offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s')))
≥ 0
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s')
{-9} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x)
⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-10} Mem?(a'_1‘type_of)
{-11} 0 ≤ a'_1‘offset
{-12} a'_1‘offset < max_linear_offset
{-13} Mem?(a'_2‘type_of)
{-14} 0 ≤ a'_2‘offset
{-15} a'_2‘offset < max_linear_offset
{-16} s'_1 > 0
{-17} s'_2 > 0
{-18} pm′‘rw_addr(a'_2)
{-19} a'_1 ≤ a'_2
{-20} a'_2 + s'_2 ≤ a'_1 + s'_1
{-21} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-22} ∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ pm′‘rw_addr(x)
{-23} address_block(data(linear_resolve(a'_2, Write)(s')), s'_2)(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s')))

{1} pm_phy‘rw_addr(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s')))

Instantiating the top quantifier in -9 with the terms: (data(linear_resolve(a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1))))), Write) (s!1))),

we get 2 subgoals:

same_page_address_block_write.1.1.1.1.1.1:

{-1} OK?(linear_resolve($a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s'))))$, Write)$(s')$)

{-2} $\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s')) + (\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
$=$
$\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s'))), \text{Write})(s'))$

{-3} $\text{rem}(\text{expt}(2, \text{ min\_page}))(\text{offset}(a_2')) + \text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2',$
$< \text{expt}(2, \text{ min\_page})$

{-4} OK?(linear_resolve(a_2', Write)(s'))

{-5} $x' =$
$\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s'))), \text{Write})(s'))$

{-6} $\text{offset}(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s'))), \text{ Write})(s'))) - \text{offset}(\text{da}$
≥ 0

{-7} is_linear_plain_memory?(pm′)

{-8} pm′‘states(s')

{-9} virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))$(\text{data}(\text{linear\_resolve}(a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s')))), \text{Write})(s'))$
$\Rightarrow$
restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)$(\text{data}(\text{linear\_resolve}(a_2' + (\text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s')))), \text{Write})(s'))$

{-10} Mem?(a_1'‘type_of)

{-11} $0 \leq a_1'$‘offset

{-12} a_1'‘offset $<$ max_linear_offset

{-13} Mem?(a_2'‘type_of)

{-14} $0 \leq a_2'$‘offset

{-15} a_2'‘offset $<$ max_linear_offset

{-16} $s_1' > 0$

{-17} $s_2' > 0$

{-18} pm′‘rw_addr(a_2')

{-19} $a_1' \leq a_2'$

{-20} $a_2' + s_2' \leq a_1' + s_1'$

{-21} $\text{rem}(\text{expt}(2, \text{ min\_page}))(\text{offset}(a_2')) + s_2' \leq \text{expt}(2, \text{ min\_page})$

{-22} $\forall$ (x: Address): address_block(a_1', s_1')(x) $\Rightarrow$ pm′‘rw_addr(x)

{-23} address_block(data(linear_resolve(a_2', Write)(s')), s_2')$(\text{data}(\text{linear\_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear\_resolve}(a_2', \text{ Write})(s'))), \text{Write})(s'))$

{1} pm_phy‘rw_addr$(\text{data}(\text{linear_resolve}(a_2' + \text{offset}(x') - \text{offset}(\text{data}(\text{linear_resolve}(a_2', \text{ Write})(s'))), \text{Write})(s'))$

Expanding the definition of restrict,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of virt_to_phys_range,

Instantiating the top quantifier in 1 with the terms: (a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1)))) Write),

we get 2 subgoals:

`same_page_address_block_write.1.1.1.1.1.1.1:`

{-1} $\mathrm{OK?}(\mathrm{linear\_resolve}(a_2' + (\mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s')))),\ \mathrm{Write})(s'))$

{-2} $\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s')) + (\mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear}$
$=$
$\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))),\ \mathrm{Write})(s'))$

{-3} $\mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))),\ \mathrm{Write})(s')))$
$< \mathrm{expt}(2,\ \mathrm{min\_page})$

{-4} $\mathrm{OK?}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))$

{-5} $x' =$
$\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))),\ \mathrm{Write})(s'))$

{-6} $\mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))),\ \mathrm{Write})(s')))$
≥ 0

{-7} $\mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}')$

{-8} $\mathrm{pm}'\text{`states}(s')$

{-9} $\mathrm{Mem?}(a_1'\text{`type\_of})$

{-10} $0 \leq a_1'\text{`offset}$

{-11} $a_1'\text{`offset} < \mathrm{max\_linear\_offset}$

{-12} $\mathrm{Mem?}(a_2'\text{`type\_of})$

{-13} $0 \leq a_2'\text{`offset}$

{-14} $a_2'\text{`offset} < \mathrm{max\_linear\_offset}$

{-15} $s_1' > 0$

{-16} $s_2' > 0$

{-17} $\mathrm{pm}'\text{`rw\_addr}(a_2')$

{-18} $a_1' \leq a_2'$

{-19} $a_2' + s_2' \leq a_1' + s_1'$

{-20} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{min\_page}))(\mathrm{offset}(a_2')) + s_2' \leq \mathrm{expt}(2,\ \mathrm{min\_page})$

{-21} $\forall\ (x\colon \mathrm{Address})\colon \mathrm{address\_block}(a_1',\ s_1')(x) \Rightarrow \mathrm{pm}'\text{`rw\_addr}(x)$

{-22} $\mathrm{address\_block}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))\ s_2')$
$(\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))),\ \mathrm{Write})(s')))$

{1} $\mathrm{OK?}(\mathrm{linear\_resolve}(a_2' + (\mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s')))),\ \mathrm{Write})(s'))$

{2} $\mathrm{pm\_phy}\text{`rw\_addr}$
$(\mathrm{data}(\mathrm{linear\_resolve}(a_2' + \mathrm{offset}(x') - \mathrm{offset}(\mathrm{data}(\mathrm{linear\_resolve}(a_2',\ \mathrm{Write})(s'))),\ \mathrm{Write})(s')))$

which is trivially true.

This completes the proof of `same_page_address_block_write.1.1.1.1.1.1.1`.

`same_page_address_block_write.1.1.1.1.1.1.2:`

{-1} OK?(linear_resolve(a_2' + (offset(x') − offset(data(linear_resolve(a_2', Write)(s')))), Write)(s'))
{-2} data(linear_resolve(a_2', Write)(s')) + (offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2',
=
data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-3} offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))) + rem(exp
< expt(2, min_page)
{-4} OK?(linear_resolve(a_2', Write)(s'))
{-5} x' =
data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-6} offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))) − offset(da
≥ 0
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s')
{-9} Mem?(a_1'‘type_of)
{-10} 0 ≤ a_1'‘offset
{-11} a_1'‘offset < max_linear_offset
{-12} Mem?(a_2'‘type_of)
{-13} 0 ≤ a_2'‘offset
{-14} a_2'‘offset < max_linear_offset
{-15} s_1' > 0
{-16} s_2' > 0
{-17} pm′‘rw_addr(a_2')
{-18} a_1' ≤ a_2'
{-19} a_2' + s_2' ≤ a_1' + s_1'
{-20} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-21} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-22} address_block(data(linear_resolve(a_2', Write)(s')), s_2')
(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s')))

{1} pm′‘rw_addr
(a_2' + (offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physical_memory, pm_phy](
{2} pm_phy‘rw_addr
(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s')))

Instantiating the top quantifier in -21 with the terms: (a2!1 + (offset(x!1) - offset(data(linear_resolve(a2!1, Write)(s!1))))),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `same_page_address_block_write.1.1.1.1.1.1.2`.

same_page_address_block_write.1.1.1.1.1.2:

{-1} OK?(linear_resolve(a'_2 + (offset(x') − offset(data(linear_resolve(a'_2, Write)(s')))), Write)(s'))
{-2} data(linear_resolve(a'_2, Write)(s')) + (offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear
= data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s'))
{-3} rem(expt(2, min_page))(offset(a'_2)) + offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear
< expt(2, min_page)
{-4} OK?(linear_resolve(a'_2, Write)(s'))
{-5} x' = data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s'))
{-6} offset(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s')))
≥ 0
{-7} is_linear_plain_memory?(pm′)
{-8} pm′‘states(s')
{-9} Mem?(a'_1‘type_of)
{-10} 0 ≤ a'_1‘offset
{-11} a'_1‘offset < max_linear_offset
{-12} Mem?(a'_2‘type_of)
{-13} 0 ≤ a'_2‘offset
{-14} a'_2‘offset < max_linear_offset
{-15} s'_1 > 0
{-16} s'_2 > 0
{-17} pm′‘rw_addr(a'_2)
{-18} a'_1 ≤ a'_2
{-19} a'_2 + s'_2 ≤ a'_1 + s'_1
{-20} rem(expt(2, min_page))(offset(a'_2)) + s'_2 ≤ expt(2, min_page)
{-21} ∀ (x: Address): address_block(a'_1, s'_1)(x) ⇒ pm′‘rw_addr(x)
{-22} address_block(data(linear_resolve(a'_2, Write)(s')), s'_2)(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s')))

{1} Mem?(data[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](
a'_2 + (offset(x') − offset(data [Physical_memory, Address](linear_resolve [Ph
Write)(s'))‘type_of)
∧
0 ≤ data[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](
a'_2 + (offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physica
Write)(s'))‘offset
∧
data[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](
a'_2 + (offset(x') − offset(data [Physical_memory, Address](linear_resolve [Physica
Write)(s'))‘offset
< max_linear_offset
{2} pm_phy‘rw_addr(data(linear_resolve(a'_2 + offset(x') − offset(data(linear_resolve(a'_2, Write)(s'))), Write)(s'))) 2108

Using lemma linear_resolve_memory_address,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `same_page_address_block_write.1.1.1.1.1.2`.
`same_page_address_block_write.1.1.1.1.2:`

{-1} data(linear_resolve(a_2', Write)(s')) + (offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2',
= data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-2} rem(expt(2, min_page))(offset(a_2')) + offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2',
< expt(2, min_page)
{-3} OK?(linear_resolve(a_2', Write)(s'))
{-4} x' = data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-5} offset(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))) − offset(da
≥ 0
{-6} is_linear_plain_memory?(pm′)
{-7} pm′‘states(s')
{-8} ∀ (x: Memory_Address_4G): virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x) ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-9} Mem?(a_1'‘type_of)
{-10} 0 ≤ a_1'‘offset
{-11} a_1'‘offset < max_linear_offset
{-12} Mem?(a_2'‘type_of)
{-13} 0 ≤ a_2'‘offset
{-14} a_2'‘offset < max_linear_offset
{-15} s_1' > 0
{-16} s_2' > 0
{-17} pm′‘rw_addr(a_2')
{-18} a_1' ≤ a_2'
{-19} a_2' + s_2' ≤ a_1' + s_1'
{-20} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-21} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-22} address_block(data(linear_resolve(a_2', Write)(s')), s_2')(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s')))

{1} OK?(linear_resolve(a_2' + (offset(x') − offset(data(linear_resolve(a_2', Write)(s')))), Write)(s'))
{2} pm_phy‘rw_addr(data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s')))

Using lemma linear_resolve_same_page_address_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `same_page_address_block_write.1.1.1.1.2`.

`same_page_address_block_write.1.1.1.2:`

{-1} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) < expt(2, min_page)
{-2} OK?(linear_resolve(a_2', Write)(s'))
{-3} data(linear_resolve(a_2', Write)(s')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) = data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-4} offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) ≥ 0
{-5} is_linear_plain_memory?(pm′)
{-6} pm′‘states(s')
{-7} ∀ (x: Memory_Address_4G): virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x) ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-8} Mem?(a_1'‘type_of)
{-9} 0 ≤ a_1'‘offset
{-10} a_1'‘offset < max_linear_offset
{-11} Mem?(a_2'‘type_of)
{-12} 0 ≤ a_2'‘offset
{-13} a_2'‘offset < max_linear_offset
{-14} s_1' > 0
{-15} s_2' > 0
{-16} pm′‘rw_addr(a_2')
{-17} a_1' ≤ a_2'
{-18} $a_2' + s_2'$ ≤ $a_1' + s_1'$
{-19} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-20} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-21} address_block(data(linear_resolve(a_2', Write)(s')), s_2')(x')

{1} data(linear_resolve(a_2', Write)(s')) + (offset(x') − offset(data(linear_resolve(a_2', Write)(s')))) = x'
{2} pm_phy‘rw_addr(x')

Expanding the definition of +,

Applying decompose-equality,

Expanding the definition of address_block,

which is trivially true.

This completes the proof of `same_page_address_block_write.1.1.1.2`.

`same_page_address_block_write.1.1.2:`

{-1} rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) < expt(2, min_page)
{-2} offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) ≥ 0
{-3} is_linear_plain_memory?(pm′)
{-4} pm′‘states(s')
{-5} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x)
⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-6} Mem?(a_1'‘type_of)
{-7} 0 ≤ a_1'‘offset
{-8} a_1'‘offset < max_linear_offset
{-9} Mem?(a_2'‘type_of)
{-10} 0 ≤ a_2'‘offset
{-11} a_2'‘offset < max_linear_offset
{-12} s_1' > 0
{-13} s_2' > 0
{-14} pm′‘rw_addr(a_2')
{-15} a_1' ≤ a_2'
{-16} $a_2' + s_2'$ ≤ $a_1' + s_1'$
{-17} rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-18} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-19} address_block(data(linear_resolve(a_2', Write)(s')), s_2')(x')

{1} OK?(linear_resolve(a_2', Write)(s'))
{2} restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_2')
{3} pm_phy‘rw_addr(x')

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `same_page_address_block_write.1.1.2`.

`same_page_address_block_write.1.2:`

{-1}	rem(expt(2, min_page))(offset(a_2')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) < expt(2, min_page) ∧ OK?(linear_resolve(a_2', Write)(s')) ⊃ data(linear_resolve(a_2', Write)(s')) + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) = data(linear_resolve(a_2' + offset(x') − offset(data(linear_resolve(a_2', Write)(s'))), Write)(s'))
{-2}	offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) ≥ 0
{-3}	is_linear_plain_memory?(pm′)
{-4}	pm′‘states(s')
{-5}	restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_2') ⊃ OK?(linear_resolve(a_2', Write)(s'))
{-6}	∀ (x: Memory_Address_4G): virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))(x) ⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-7}	Mem?(a_1'‘type_of)
{-8}	0 ≤ a_1'‘offset
{-9}	a_1'‘offset < max_linear_offset
{-10}	Mem?(a_2'‘type_of)
{-11}	0 ≤ a_2'‘offset
{-12}	a_2'‘offset < max_linear_offset
{-13}	s_1' > 0
{-14}	s_2' > 0
{-15}	pm′‘rw_addr(a_2')
{-16}	a_1' ≤ a_2'
{-17}	a_2' + s_2' ≤ a_1' + s_1'
{-18}	rem(expt(2, min_page))(offset(a_2')) + s_2' ≤ expt(2, min_page)
{-19}	∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-20}	address_block(data(linear_resolve(a_2', Write)(s')), s_2')(x')
{1}	rem(expt(2, min_page))(offset(a_2')) + (offset(x') − offset(data(linear_resolve(a_2', Write)(s')))) < expt(2, min_page)
{2}	pm_phy‘rw_addr(x')

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `same_page_address_block_write.1.2`.

`same_page_address_block_write.2:`

{-1} is_linear_plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(a_2') ⊃
OK?(linear_resolve(a_2', Write)(s'))
{-4} ∀ (x: Memory_Address_4G):
virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr),
singleton(Write))
(x)
⇒ restrict[Address, Memory_Address_4G, boolean](pm_phy‘rw_addr)(x)
{-5} Mem?(a_1'‘type_of)
{-6} $0 \leq a_1'$‘offset
{-7} a_1'‘offset < max_linear_offset
{-8} Mem?(a_2'‘type_of)
{-9} $0 \leq a_2'$‘offset
{-10} a_2'‘offset < max_linear_offset
{-11} $s_1' > 0$
{-12} $s_2' > 0$
{-13} pm′‘rw_addr(a_2')
{-14} $a_1' \leq a_2'$
{-15} $a_2' + s_2' \leq a_1' + s_1'$
{-16} rem(expt(2, min_page))(offset(a_2')) + $s_2' \leq$ expt(2, min_page)
{-17} ∀ (x: Address): address_block(a_1', s_1')(x) ⇒ pm′‘rw_addr(x)
{-18} address_block(data(linear_resolve(a_2', Write)(s')), s_2')(x')

{1} offset(x') − offset(data(linear_resolve(a_2', Write)(s'))) ≥ 0
{2} pm_phy‘rw_addr(x')

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `same_page_address_block_write.2`.
Q.E.D.

C.117.74 Linear_Memory_Properties.split_linear_read_side_effects_states_TCC1

Terse proof for `split_linear_read_side_effects_states_TCC1`.

`split_linear_read_side_effects_states_TCC1:`

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])):
is_linear_plain_memory?(pm) ∧
(address_block(a, length(bl)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
⊃
every[[Address, list[Byte]]]
(λ (t: [Address, list[Byte]]):
Mem?(t‘1‘type_of) ∧ $0 \leq t$‘1‘offset ∧ t‘1‘offset < max_linear_offset)
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Using lemma address_block_split_type,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_read_side_effects_states_TCC1`.
Q.E.D.

C.117.75 Linear_Memory_Properties.split_linear_read_side_effects_states

Terse proof for `split_linear_read_side_effects_states`.

`split_linear_read_side_effects_states:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])):
       is_linear_plain_memory?(pm) ∧
        (address_block(a, length(bl)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
        ⊃
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                  transformer_invariant?(pm‘states,
                                         singleton(expr_2_super(linear_read_side_effect_in_page
                                                                (e)))))
             (split(min_page, a, bl))
```

Repeatedly Skolemizing and flattening,
Using lemma split_range,
Using lemma split_no_null,
Using lemma split_type,
Using lemma split_pair_cross_size,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`split_linear_read_side_effects_states.1:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                type_of(t‘1) = type_of(a′) ∧
                 reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
                  t‘1 < reg_size(max_linear)(type_of(a′)))
           (split(min_page, a′, bl′))
{-3}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-4}  every(λ (a2: Address, bl2: list[Byte]):
                a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
           (split(min_page, a′, bl′))
{-5}  Mem?(a′‘type_of)
{-6}  0 ≤ a′‘offset
{-7}  a′‘offset < max_linear_offset
{-8}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl′)
{-9}  cons?[Byte](bl′)
{-10} is_linear_plain_memory?(pm′)
{-11} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
------------------------------------------------------------------------
 {1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′‘states,
                                       singleton(expr_2_super(linear_read_side_effect_in_page(e))))
           (split(min_page, a′, bl′))
```

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

`split_linear_read_side_effects_states.1.1:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e))))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_states.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \le$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\le a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\le$ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) $\le$
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \le t$‘1 ∧
t‘1 + length(t‘2) $\le a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\le t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) $\le$
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \le t$‘1 ∧ t‘1 + length(t‘2) $\le a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\le t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \le$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\le a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\le$ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \le a_2$ ∧ a_2 + length(bl2) $\le a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \le t$‘1 ∧ t‘1 + length(t‘2) $\le a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) $\le$ expt(2, min_page))
(split(min_page, a', bl′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\le t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),

Installing automatic rewrites from: max_linear Mem min_linear

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

Using lemma every_implied,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of transformer_invariant?,

Expanding the definition of linear_read_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -9,

Hiding formulas: -9,

Installing automatic rewrites from: (result_pred! has_next_state! ##! expr_2_super! expr_2_super_res!)

Case splitting on union(pm!1`ro_addr, pm!1`rw_addr)(t!1`1),

we get 2 subgoals:

`split_linear_read_side_effects_states.1.1.1.1:`

{-1} union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-2} cons?(t′`2)
{-3} $a' \leq t'`1$
{-4} $t'`1 + \text{length}(t'`2) \leq a' + \text{length}(\text{bl}')$
{-5} type_of(t′`1) = type_of(a')
{-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t′`1
{-7} $t'`1 <$ reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-8} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-9} pm′`states(s')
{-10} Mem?(a'`type_of)
{-11} $0 \leq a'$`offset
{-12} a'`offset < max_linear_offset
{-13} cons?[Byte](bl′)
{-14} is_linear_plain_memory?(pm′)
{-15} (address_block(a', length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))

{1} result_pred(pm′`states)
(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
(linear_resolve(t′`1, Read) ##
(λ (pa: Memory_Address_4G)
(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy`mem)
(pa, t′`2, TRUE)(s)))
(s))
(s′))

Using lemma pm_linear_resolve_read_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1`1, Read)(s!1)), length(t!1`2)), union(pm_phy`ro_addr, pm_phy`rw_addr)),

we get 3 subgoals:

`split_linear_read_side_effects_states.1.1.1.1.1:`

{-1} (address_block(data(linear_resolve(t'‘1, Read)(s')), length(t'‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw
{-2} is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'‘1, Read)(s'))
{-3} union(pm'‘ro_addr, pm'‘rw_addr)(t'‘1)
{-4} cons?(t'‘2)
{-5} a' ≤ t'‘1
{-6} t'‘1 + length(t'‘2) ≤ a' + length(bl')
{-7} type_of(t'‘1) = type_of(a')
{-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'‘1
{-9} t'‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10} rem(expt(2, min_page))(offset(t'‘1)) + length(t'‘2) ≤ expt(2, min_page)
{-11} pm'‘states(s')
{-12} Mem?(a'‘type_of)
{-13} 0 ≤ a'‘offset
{-14} a'‘offset < max_linear_offset
{-15} cons?[Byte](bl')
{-16} is_linear_plain_memory?(pm')
{-17} (address_block(a', length(bl')) ⊆ (pm'‘ro_addr ∪ pm'‘rw_addr))

{1} result_pred(pm'‘states)
(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
(linear_resolve(t'‘1, Read) ##
(λ (pa: Memory_Address_4G)
(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)
(pa, t'‘2, TRUE)(s)))
(s))
(s'))

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`split_linear_read_side_effects_states.1.1.1.1.1.1:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′`states = pm_phy`states
{-4}	union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-5}	pm′`states(s′)
{-6}	OK?(linear_resolve(t′`1, Read)(s′))
{-7}	pm_phy`states(state(linear_resolve(t′`1, Read)(s′)))
{-8}	(address_block(data(linear_resolve(t′`1, Read)(s′)), length(t′`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-9}	cons?(t′`2)
{-10}	a′ ≤ t′`1
{-11}	t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-12}	type_of(t′`1) = type_of(a′)
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-14}	t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-15}	rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-16}	OK?(memory_read_side_effect(pm_phy`mem) (data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)(s′))
{-17}	Mem?(a′`type_of)
{-18}	0 ≤ a′`offset
{-19}	a′`offset < max_linear_offset
{-20}	cons?[Byte](bl′)
{-21}	(address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{1}	pm′`states (state(memory_read_side_effect(pm_phy`mem) (data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)(s′)))

Using lemma plain_memory_transformer_invariant_read_side_effects,

Using lemma super_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating the top quantifier in 1 with the terms: (data(linear_resolve(t!1`1, Read)(s!1)) t!1`2 TRUE),

Rewriting using subset_reflexive, matching in *,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.1.1`.

`split_linear_read_side_effects_states.1.1.1.1.1.2:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′‘states = pm_phy‘states
{-4}	union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-5}	pm′‘states(s′)
{-6}	OK?(linear_resolve(t′‘1, Read)(s′))
{-7}	pm_phy‘states(state(linear_resolve(t′‘1, Read)(s′)))
{-8}	(address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw
{-9}	cons?(t′‘2)
{-10}	a′ ≤ t′‘1
{-11}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-12}	type_of(t′‘1) = type_of(a′)
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-14}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-15}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-16}	Exception?(memory_read_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Read)(s′)), t′‘2, TRUE)(s′))
{-17}	Mem?(a′‘type_of)
{-18}	0 ≤ a′‘offset
{-19}	a′‘offset < max_linear_offset
{-20}	cons?[Byte](bl′)
{-21}	(address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{1}	pm′‘states (state(memory_read_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Read)(s′)), t′‘2, TRUE)(s′)))

Using lemma plain_memory_transformers_ok_read_side_effects_block,

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating the top quantifier in 1 with the terms: (data(linear_resolve(t!1‘1, Read)(s!1)) t!1‘2 TRUE),

Rewriting using subset_reflexive, matching in *,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.1.2`.

`split_linear_read_side_effects_states.1.1.1.1.2:`

```
  {-1}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'`1, Read)(s'))
  {-2}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
  {-3}  cons?(t'`2)
  {-4}  a' ≤ t'`1
  {-5}  t'`1 + length(t'`2) ≤ a' + length(bl')
  {-6}  type_of(t'`1) = type_of(a')
  {-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
  {-8}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
  {-9}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
 {-10}  pm'`states(s')
 {-11}  Mem?(a'`type_of)
 {-12}  0 ≤ a'`offset
 {-13}  a'`offset < max_linear_offset
 {-14}  cons?[Byte](bl')
 {-15}  is_linear_plain_memory?(pm')
 {-16}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
 ------------------------------------------------------------------------
  {1}   (address_block(data(linear_resolve(t'`1, Read)(s')), length(t'`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
  {2}   result_pred(pm'`states)
                  (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                  (linear_resolve(t'`1, Read) ##
                                     (λ (pa: Memory_Address_4G)
                                        (ns: Linear_memory[Physical_memory, pm_phy]):
                                        memory_read_side_effect(pm_phy`mem)
                                           (pa, t'`2, TRUE)(s)))
                                   (s))
                                (s'))
```

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

`split_linear_read_side_effects_states.1.1.1.1.2.1:`

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1) ∧
a′ ≤ t′‘1 ∧
t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧
rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
∧ (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
⊃
(address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘r
{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-4} cons?(t′‘2)
{-5} a′ ≤ t′‘1
{-6} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7} type_of(t′‘1) = type_of(a′)
{-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11} pm′‘states(s′)
{-12} Mem?(a′‘type_of)
{-13} 0 ≤ a′‘offset
{-14} a′‘offset < max_linear_offset
{-15} cons?[Byte](bl′)
{-16} is_linear_plain_memory?(pm′)
{-17} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.2.1`.
`split_linear_read_side_effects_states.1.1.1.1.2.2:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} pm′‘states(s′)
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} length[Byte](t′‘2) > 0
{2} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.2.2`.

`split_linear_read_side_effects_states.1.1.1.1.2.3`:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Read)(s′))
{-2}  union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-3}  cons?(t′`2)
{-4}  a′ ≤ t′`1
{-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-6}  type_of(t′`1) = type_of(a′)
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-10} pm′`states(s′)
{-11} Mem?(a′`type_of)
{-12} 0 ≤ a′`offset
{-13} a′`offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
 {1}  length[Byte](bl′) > 0
 {2}  (address_block(data(linear_resolve(t′`1, Read)(s′)), length(t′`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.2.3`.

`split_linear_read_side_effects_states.1.1.1.1.3`:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Read)(s′))
{-2}  union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-3}  cons?(t′`2)
{-4}  a′ ≤ t′`1
{-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-6}  type_of(t′`1) = type_of(a′)
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-10} pm′`states(s′)
{-11} Mem?(a′`type_of)
{-12} 0 ≤ a′`offset
{-13} a′`offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
 {1}  OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t′`1, Read)(s′))
 {2}  result_pred(pm′`states)
                 (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                 (linear_resolve(t′`1, Read) ##
                                    (λ (pa: Memory_Address_4G)
                                       (ns: Linear_memory[Physical_memory, pm_phy]):
                                       memory_read_side_effect(pm_phy`mem)
                                          (pa, t′`2, TRUE)(s)))
                                  (s))
                               (s′))
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.1.3`.

`split_linear_read_side_effects_states.1.1.1.2`:

```
{-1}   cons?(t'`2)
{-2}   a' ≤ t'`1
{-3}   t'`1 + length(t'`2) ≤ a' + length(bl')
{-4}   type_of(t'`1) = type_of(a')
{-5}   reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-6}   t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}   rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-8}   pm'`states(s')
{-9}   Mem?(a'`type_of)
{-10}  0 ≤ a'`offset
{-11}  a'`offset < max_linear_offset
{-12}  cons?[Byte](bl')
{-13}  is_linear_plain_memory?(pm')
{-14}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
-------------------------------------------------------------------------------
{1}    union(pm'`ro_addr, pm'`rw_addr)(t'`1)
{2}    result_pred(pm'`states)
         (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         (linear_resolve(t'`1, Read) ##
                            (λ (pa: Memory_Address_4G)
                                 (ns: Linear_memory[Physical_memory, pm_phy]):
                               memory_read_side_effect(pm_phy`mem)
                                  (pa, t'`2, TRUE)(s)))
                          (s))
                    (s'))
```

Hiding formulas: (-7 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.1.1.1.2`.

split_linear_read_side_effects_states.1.1.2:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} ∀ (t_1: [Address, list[Byte]]):
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(max_linear‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.1.2`.

split_linear_read_side_effects_states.1.1.3:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(min_linear‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.1.3`.

`split_linear_read_side_effects_states.1.2:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t‘2) ∧
                 a' ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
                type_of(t‘1) = type_of(a') ∧
                 reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
                  t‘1 < reg_size(max_linear)(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'‘type_of)
{-7}  0 ≤ a'‘offset
{-8}  a'‘offset < max_linear_offset
{-9}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ (pm'‘ro_addr ∪ pm'‘rw_addr))
------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧ type_of(t‘1) = type_of(a') ⊃
         Mem?(max_linear‘type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm'‘states,
                                       singleton(expr_2_super(linear_read_side_effect_in_page(e)))
           (split(min_page, a', bl'))
```

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.2`.

`split_linear_read_side_effects_states.1.3:`

{-1} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-2} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
{-4} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-5} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-6} Mem?(a′‘type_of)
{-7} 0 ≤ a′‘offset
{-8} a′‘offset < max_linear_offset
{-9} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-10} cons?[Byte](bl′)
{-11} is_linear_plain_memory?(pm′)
{-12} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} ∀ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ⊃ Mem?(min_linear‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_states.1.3`.

`split_linear_read_side_effects_states.2:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
            (split(min_page, a', bl'))
{-4}  Mem?(a'`type_of)
{-5}  0 ≤ a'`offset
{-6}  a'`offset < max_linear_offset
{-7}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl')
{-8}  cons?[Byte](bl')
{-9}  is_linear_plain_memory?(pm')
{-10} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
-------------------------------------------------------------------------
 {1}  a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm'`states,
                                       singleton(expr_2_super(linear_read_side_effect_in_page(e)))
            (split(min_page, a', bl'))
```

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: Mem max_linear bus_width min_linear

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_states.2`.

`split_linear_read_side_effects_states.3:`

{-1} every(λ (a₂: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3} every(λ (a₂: Address, bl2: list[Byte]):
a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-4} Mem?(a′`type_of)
{-5} 0 ≤ a′`offset
{-6} a′`offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-8} cons?[Byte](bl′)
{-9} is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))

{1} reg_base(min_linear)(type_of(a′)) ≤ a′
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′`states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Expanding the definition of reg_base,
Expanding the definition of min_linear,
Expanding the definition of Mem,
Expanding the definition of <=,
which is trivially true.
This completes the proof of `split_linear_read_side_effects_states.3`.
Q.E.D.

C.117.76 Linear_Memory_Properties.split_linear_write_side_effects_states_TCC1

Terse proof for `split_linear_write_side_effects_states_TCC1`.

`split_linear_write_side_effects_states_TCC1:`

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])):
is_linear_plain_memory?(pm) ∧ (address_block(a, length(bl)) ⊆ pm`rw_addr) ⊃
every[[Address, list[Byte]]]
(λ (t: [Address, list[Byte]]):
Mem?(t`1`type_of) ∧ 0 ≤ t`1`offset ∧ t`1`offset < max_linear_offset)
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Using lemma address_block_split_type,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-7 1) and hiding *,
Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets pm!1`rw_addr, c gets union(pm!1`ro_addr, pm!1`rw_addr),

Keeping (1) and hiding *,

Rewriting using union_commutative, matching in *,

Rewriting using union_subset1, matching in *,

This completes the proof of `split_linear_write_side_effects_states_TCC1`.

Q.E.D.

C.117.77 Linear_Memory_Properties.split_linear_write_side_effects_states

Terse proof for `split_linear_write_side_effects_states`.

`split_linear_write_side_effects_states:`

```
{1}  ∀ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])):
       is_linear_plain_memory?(pm) ∧ (address_block(a, length(bl)) ⊆ pm`rw_addr) ⊃
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm`states,
                                       singleton(expr_2_super(linear_write_side_effect_in_page
                                                              (e)))))
              (split(min_page, a, bl))
```

Repeatedly Skolemizing and flattening,

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

split_linear_write_side_effects_states.1:

{-1} every(λ (a₂: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
{-3} every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-4} every(λ (a₂: Address, bl2: list[Byte]):
a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-5} Mem?(a′‘type_of)
{-6} 0 ≤ a′‘offset
{-7} a′‘offset < max_linear_offset
{-8} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-9} cons?[Byte](bl′)
{-10} is_linear_plain_memory?(pm′)
{-11} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

`split_linear_write_side_effects_states.1.1:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_write_side_effect_in_page(e))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_write_side_effects_states.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2 \wedge a_2$ + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a', bl′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),

Installing automatic rewrites from: (min_linear! max_linear! Mem!)

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

Using lemma every_implied,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of transformer_invariant?,

Expanding the definition of linear_write_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -9,

Hiding formulas: -9,

Installing automatic rewrites from: (result_pred! has_next_state! ##! expr_2_super! expr_2_super_res!)

Case splitting on pm!1‘rw_addr(t!1‘1),

we get 2 subgoals:

`split_linear_write_side_effects_states.1.1.1.1:`

{-1} $pm'\text{`}rw\_addr(t'\text{`}1)$
{-2} $cons?(t'\text{`}2)$
{-3} $a' \leq t'\text{`}1$
{-4} $t'\text{`}1 + length(t'\text{`}2) \leq a' + length(bl')$
{-5} $type\_of(t'\text{`}1) = type\_of(a')$
{-6} $reg\_base((\#type\_of := Mem\_,\ offset := 0\#))(type\_of(a')) \leq t'\text{`}1$
{-7} $t'\text{`}1 < reg\_size((\#type\_of := Mem\_,\ offset := max\_linear\_offset\#))(type\_of(a'))$
{-8} $rem(expt(2,\ min\_page))(offset(t'\text{`}1)) + length(t'\text{`}2) \leq expt(2,\ min\_page)$
{-9} $pm'\text{`}states(s')$
{-10} $Mem?(a'\text{`}type\_of)$
{-11} $0 \leq a'\text{`}offset$
{-12} $a'\text{`}offset < max\_linear\_offset$
{-13} $cons?[Byte](bl')$
{-14} $is\_linear\_plain\_memory?(pm')$
{-15} $(address\_block(a',\ length(bl')) \subseteq pm'\text{`}rw\_addr)$

{1}
```
result_pred(pm'`states)
                    (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                     (linear_resolve(t'`1, Write) ##
                                        (λ (pa: Memory_Address_4G)
                                              (ns: Linear_memory[Physical_memory, pm_phy]):
                                            memory_write_side_effect(pm_phy`mem)
                                                (pa, t'`2, TRUE)(s)))
                                       (s))
                                  (s'))
```

Using lemma pm_linear_resolve_write_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1‘1, Write)(s!1)), length(t!1‘2)), pm_phy‘rw_addr),

we get 3 subgoals:

`split_linear_write_side_effects_states.1.1.1.1.1:`

```
{-1}  (address_block(data(linear_resolve(t'‘1, Write)(s')), length(t'‘2)) ⊆ pm_phy‘rw_addr)
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'‘1, Write)(s'))
{-3}  pm'‘rw_addr(t'‘1)
{-4}  cons?(t'‘2)
{-5}  a' ≤ t'‘1
{-6}  t'‘1 + length(t'‘2) ≤ a' + length(bl')
{-7}  type_of(t'‘1) = type_of(a')
{-8}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'‘1
{-9}  t'‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10} rem(expt(2, min_page))(offset(t'‘1)) + length(t'‘2) ≤ expt(2, min_page)
{-11} pm'‘states(s')
{-12} Mem?(a'‘type_of)
{-13} 0 ≤ a'‘offset
{-14} a'‘offset < max_linear_offset
{-15} cons?[Byte](bl')
{-16} is_linear_plain_memory?(pm')
{-17} (address_block(a', length(bl')) ⊆ pm'‘rw_addr)
-------------------------------------------------------------------------------
 {1}  result_pred(pm'‘states)
                (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                (linear_resolve(t'‘1, Write) ##
                                   (λ (pa: Memory_Address_4G)
                                      (ns: Linear_memory[Physical_memory, pm_phy]):
                                      memory_write_side_effect(pm_phy‘mem)
                                         (pa, t'‘2, TRUE)(s)))
                                 (s))
                              (s'))
```

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

`split_linear_write_side_effects_states.1.1.1.1.1.1:`

{-1} is_linear_plain_memory?(pm′)
{-2} plain_memory?(pm_phy)
{-3} pm′‘states = pm_phy‘states
{-4} pm′‘states(s′)
{-5} OK?(linear_resolve(t′‘1, Write)(s′))
{-6} pm_phy‘states(state(linear_resolve(t′‘1, Write)(s′)))
{-7} (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-8} pm′‘rw_addr(t′‘1)
{-9} cons?(t′‘2)
{-10} a′ ≤ t′‘1
{-11} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-12} type_of(t′‘1) = type_of(a′)
{-13} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-14} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-15} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-16} OK?(memory_write_side_effect(pm_phy‘mem)
(data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))
{-17} Mem?(a′‘type_of)
{-18} 0 ≤ a′‘offset
{-19} a′‘offset < max_linear_offset
{-20} cons?[Byte](bl′)
{-21} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{1} pm′‘states
(state(memory_write_side_effect(pm_phy‘mem)
(data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′)))

Using lemma plain_memory_transformer_invariant_write_side_effects,

Using lemma super_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating the top quantifier in 1 with the terms: (data(linear_resolve(t!1‘1, Write)(s!1)) t!1‘2 TRUE),

Rewriting using subset_reflexive, matching in *,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.1.1`.

`split_linear_write_side_effects_states.1.1.1.1.1.2:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′‘states = pm_phy‘states
{-4}	pm′‘states(s′)
{-5}	OK?(linear_resolve(t′‘1, Write)(s′))
{-6}	pm_phy‘states(state(linear_resolve(t′‘1, Write)(s′)))
{-7}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-8}	pm′‘rw_addr(t′‘1)
{-9}	cons?(t′‘2)
{-10}	a′ ≤ t′‘1
{-11}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-12}	type_of(t′‘1) = type_of(a′)
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-14}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-15}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-16}	Exception?(memory_write_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))
{-17}	Mem?(a′‘type_of)
{-18}	0 ≤ a′‘offset
{-19}	a′‘offset < max_linear_offset
{-20}	cons?[Byte](bl′)
{-21}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{1}	pm′‘states (state(memory_write_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′)))

Using lemma plain_memory_transformers_ok_write_side_effects_block,

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating the top quantifier in 1 with the terms: (data(linear_resolve(t!1‘1, Write)(s!1)) t!1‘2 TRUE),

Rewriting using subset_reflexive, matching in *,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.1.2`.

`split_linear_write_side_effects_states.1.1.1.1.1.3:`

{-1} is_linear_plain_memory?(pm′)
{-2} plain_memory?(pm_phy)
{-3} pm′`states = pm_phy`states
{-4} pm′`states(s′)
{-5} OK?(linear_resolve(t′`1, Write)(s′))
{-6} (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
{-7} pm′`rw_addr(t′`1)
{-8} cons?(t′`2)
{-9} a′ ≤ t′`1
{-10} t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-11} type_of(t′`1) = type_of(a′)
{-12} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-13} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-14} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-15} OK?(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′))
{-16} Mem?(a′`type_of)
{-17} 0 ≤ a′`offset
{-18} a′`offset < max_linear_offset
{-19} cons?[Byte](bl′)
{-20} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)

{1} union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{2} pm′`states
(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′)))

Keeping (-7 1) and hiding *,

Rewriting using union_right, matching in *,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.1.3`.

`split_linear_write_side_effects_states.1.1.1.1.1.4:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′‘states = pm_phy‘states
{-4}	pm′‘states(s′)
{-5}	OK?(linear_resolve(t′‘1, Write)(s′))
{-6}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-7}	pm′‘rw_addr(t′‘1)
{-8}	cons?(t′‘2)
{-9}	a′ ≤ t′‘1
{-10}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-11}	type_of(t′‘1) = type_of(a′)
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-13}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-14}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-15}	Exception?(memory_write_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))
{-16}	Mem?(a′‘type_of)
{-17}	0 ≤ a′‘offset
{-18}	a′‘offset < max_linear_offset
{-19}	cons?[Byte](bl′)
{-20}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{1}	union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{2}	pm′‘states (state(memory_write_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′)))

Rewriting using union_right, matching in *,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.1.4`.

`split_linear_write_side_effects_states.1.1.1.1.2:`

```
  {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
  {-2}  pm′`rw_addr(t′`1)
  {-3}  cons?(t′`2)
  {-4}  a′ ≤ t′`1
  {-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
  {-6}  type_of(t′`1) = type_of(a′)
  {-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
  {-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
 {-10}  pm′`states(s′)
 {-11}  Mem?(a′`type_of)
 {-12}  0 ≤ a′`offset
 {-13}  a′`offset < max_linear_offset
 {-14}  cons?[Byte](bl′)
 {-15}  is_linear_plain_memory?(pm′)
 {-16}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
--------------------------------------------------------------------------------
  {1}   (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
  {2}   result_pred(pm′`states)
                  (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                  (linear_resolve(t′`1, Write) ##
                                     (λ (pa: Memory_Address_4G)
                                        (ns: Linear_memory[Physical_memory, pm_phy]):
                                        memory_write_side_effect(pm_phy`mem)
                                           (pa, t′`2, TRUE)(s)))
                                   (s))
                                (s′))
```

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

`split_linear_write_side_effects_states.1.1.1.1.2.1:`

{-1}	is_linear_plain_memory?(pm′) ∧ pm′‘states(s′) ∧ pm′‘rw_addr(t′‘1) ∧ a′ ≤ t′‘1 ∧ t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧ rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page) ∧ (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr) ⊃ (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-2}	is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-3}	pm′‘rw_addr(t′‘1)
{-4}	cons?(t′‘2)
{-5}	a′ ≤ t′‘1
{-6}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7}	type_of(t′‘1) = type_of(a′)
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11}	pm′‘states(s′)
{-12}	Mem?(a′‘type_of)
{-13}	0 ≤ a′‘offset
{-14}	a′‘offset < max_linear_offset
{-15}	cons?[Byte](bl′)
{-16}	is_linear_plain_memory?(pm′)
{-17}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{1}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.2.1`.
`split_linear_write_side_effects_states.1.1.1.1.2.2:`

{-1}	is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2}	pm′‘rw_addr(t′‘1)
{-3}	cons?(t′‘2)
{-4}	a′ ≤ t′‘1
{-5}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6}	type_of(t′‘1) = type_of(a′)
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10}	pm′‘states(s′)
{-11}	Mem?(a′‘type_of)
{-12}	0 ≤ a′‘offset
{-13}	a′‘offset < max_linear_offset
{-14}	cons?[Byte](bl′)
{-15}	is_linear_plain_memory?(pm′)
{-16}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{1}	length[Byte](t′‘2) > 0
{2}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.2.2`.
`split_linear_write_side_effects_states.1.1.1.1.2.3`:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
{-2}  pm′`rw_addr(t′`1)
{-3}  cons?(t′`2)
{-4}  a′ ≤ t′`1
{-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-6}  type_of(t′`1) = type_of(a′)
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-10} pm′`states(s′)
{-11} Mem?(a′`type_of)
{-12} 0 ≤ a′`offset
{-13} a′`offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
-------------------------------------------------------------------------
 {1}  length[Byte](bl′) > 0
 {2}  (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
```

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.2.3`.
`split_linear_write_side_effects_states.1.1.1.1.3`:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
{-2}  pm′`rw_addr(t′`1)
{-3}  cons?(t′`2)
{-4}  a′ ≤ t′`1
{-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-6}  type_of(t′`1) = type_of(a′)
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-10} pm′`states(s′)
{-11} Mem?(a′`type_of)
{-12} 0 ≤ a′`offset
{-13} a′`offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
-------------------------------------------------------------------------
 {1}  OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t′`1, Write)(s′))
 {2}  result_pred(pm′`states)
                (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                (linear_resolve(t′`1, Write) ##
                                   (λ (pa: Memory_Address_4G)
                                      (ns: Linear_memory[Physical_memory, pm_phy]):
                                     memory_write_side_effect(pm_phy`mem)
                                        (pa, t′`2, TRUE)(s)))
                                 (s))
                           (s′))
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.1.3`.

`split_linear_write_side_effects_states.1.1.1.2`:

```
  {-1}   cons?(t'`2)
  {-2}   a' ≤ t'`1
  {-3}   t'`1 + length(t'`2) ≤ a' + length(bl')
  {-4}   type_of(t'`1) = type_of(a')
  {-5}   reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
  {-6}   t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
  {-7}   rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
  {-8}   pm'`states(s')
  {-9}   Mem?(a'`type_of)
  {-10}  0 ≤ a'`offset
  {-11}  a'`offset < max_linear_offset
  {-12}  cons?[Byte](bl')
  {-13}  is_linear_plain_memory?(pm')
  {-14}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
  ------------------------------------------------------------------------------
  {1}    pm'`rw_addr(t'`1)
  {2}    result_pred(pm'`states)
                    (expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                    (linear_resolve(t'`1, Write) ##
                                       (λ (pa: Memory_Address_4G)
                                          (ns: Linear_memory[Physical_memory, pm_phy]):
                                          memory_write_side_effect(pm_phy`mem)
                                             (pa, t'`2, TRUE)(s)))
                                     (s))
                                (s'))
```

Hiding formulas: (-7 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.1.1.1.2`.

split_linear_write_side_effects_states.1.1.2:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{1} ∀ (t_1: [Address, list[Byte]]):
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(max_linear‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.1.2`.

split_linear_write_side_effects_states.1.1.3:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(min_linear‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
(split(min_page, a′, bl′))

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.1.3`.

`split_linear_write_side_effects_states.1.2:`

```
{-1}  every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
       every(λ (a₂: Address, bl2: list[Byte]):
                a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
             (split(min_page, a′, bl′))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t‘2) ∧
                 a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
             (split(min_page, a′, bl′))
{-2}  every(λ (a₂: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
             (split(min_page, a′, bl′))
{-3}  every(λ (t: [Address, list[Byte]]):
                type_of(t‘1) = type_of(a′) ∧
                 reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
                  t‘1 < reg_size(max_linear)(type_of(a′)))
             (split(min_page, a′, bl′))
{-4}  every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-5}  every(λ (a₂: Address, bl2: list[Byte]):
                a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
             (split(min_page, a′, bl′))
{-6}  Mem?(a′‘type_of)
{-7}  0 ≤ a′‘offset
{-8}  a′‘offset < max_linear_offset
{-9}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
             (bl′)
{-10} cons?[Byte](bl′)
{-11} is_linear_plain_memory?(pm′)
{-12} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧ type_of(t‘1) = type_of(a′) ⊃
         Mem?(max_linear‘type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′‘states,
                                       singleton(expr_2_super(linear_write_side_effect_in_page(e))
             (split(min_page, a′, bl′))
```

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.2`.

`split_linear_write_side_effects_states.1.3:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                 a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                 cons?(t`2) ∧
                  a' ≤ t`1 ∧ t`1 + length(t`2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
                type_of(t`1) = type_of(a') ∧
                 reg_base(min_linear)(type_of(a')) ≤ t`1 ∧
                  t`1 < reg_size(max_linear)(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'`type_of)
{-7}  0 ≤ a'`offset
{-8}  a'`offset < max_linear_offset
{-9}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
-------------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        type_of(t`1) = type_of(a') ⊃ Mem?(min_linear`type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm'`states,
                                       singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
           (split(min_page, a', bl'))
```

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_write_side_effects_states.1.3`.

`split_linear_write_side_effects_states.2:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
            (split(min_page, a', bl'))
{-4}  Mem?(a'`type_of)
{-5}  0 ≤ a'`offset
{-6}  a'`offset < max_linear_offset
{-7}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl')
{-8}  cons?[Byte](bl')
{-9}  is_linear_plain_memory?(pm')
{-10} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
-------------------------------------------------------------------------------
 {1}  a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm'`states,
                                       singleton(expr_2_super(linear_write_side_effect_in_page(e))
            (split(min_page, a', bl'))
```

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: Mem max_linear bus_width min_linear

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_states.2`.

`split_linear_write_side_effects_states.3:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a′, bl′))
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
            (split(min_page, a′, bl′))
{-4}  Mem?(a′`type_of)
{-5}  0 ≤ a′`offset
{-6}  a′`offset < max_linear_offset
{-7}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-8}  cons?[Byte](bl′)
{-9}  is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
------------------------------------------------------------------------------
 {1}  reg_base(min_linear)(type_of(a′)) ≤ a′
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′`states,
                                       singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
            (split(min_page, a′, bl′))
```

Expanding the definition of reg_base,
Expanding the definition of min_linear,
Expanding the definition of Mem,
Expanding the definition of <=,
which is trivially true.
This completes the proof of `split_linear_write_side_effects_states.3`.
Q.E.D.

C.117.78 Linear_Memory_Properties.split_linear_read_side_effects_ok

Terse proof for `split_linear_read_side_effects_ok`.

`split_linear_read_side_effects_ok:`

```
------------------------------------------------------------------------------
{1}  ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
          a: Memory_Address_4G, bl: (cons?[Byte])):
       is_linear_plain_memory?(pm) ∧
        (address_block(a, length(bl)) ⊆ (pm`ro_addr ∪ pm`rw_addr)) ∧ pm`states(s)
        ⊃
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                  OK?(linear_read_side_effect_in_page(e)(s)))
            (split(min_page, a, bl))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: Mem!! max_linear!! min_linear!!
Using lemma split_range,
Using lemma split_no_null,
Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_read_side_effects_ok.1:`

{-1} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))
{-3} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-4} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-5} Mem?(a′‘type_of)
{-6} 0 ≤ a′‘offset
{-7} a′‘offset < max_linear_offset
{-8} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-9} cons?[Byte](bl′)
{-10} is_linear_plain_memory?(pm′)
{-11} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-12} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-14} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq$ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) $\leq$
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) $\leq$
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq$ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2$ ∧ a_2 + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) $\leq$ expt(2, min_page))
(split(min_page, a', bl′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t$‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
(split(min_page, a', bl′))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),
Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
we get 3 subgoals:
`split_linear_read_side_effects_ok.1.1.1.1:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
$a' \leq$ s‘1 ∧
s‘1 + length(s‘2) $\leq a'$ + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq$ t‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-3} Mem?(a'‘type_of)
{-4} 0 ≤ a'‘offset
{-5} a'‘offset < max_linear_offset
{-6} cons?[Byte](bl′)
{-7} is_linear_plain_memory?(pm′)
{-8} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-9} pm′‘states(s')

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s')))
(split(min_page, a', bl′))

Using lemma every_implied,
we get 3 subgoals:

split_linear_read_side_effects_ok.1.1.1.1.1:

{-1} (∀ (t: [Memory_Address_4G, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page)
⊃ OK?(linear_read_side_effect_in_page(t)(s′)))
∧
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} cons?[Byte](bl′)
{-8} is_linear_plain_memory?(pm′)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_read_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on union(pm!1‘ro_addr, pm!1‘rw_addr)(t!1‘1),

we get 2 subgoals:

`split_linear_read_side_effects_ok.1.1.1.1.1.1:`

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-2} cons?(t′‘2)
{-3} $a' \leq t'‘1$
{-4} $t'‘1 + \text{length}(t'‘2) \leq a' + \text{length}(\text{bl}')$
{-5} type_of(t′‘1) = type_of(a′)
{-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-7} $t'‘1 < \text{reg\_size}((\#\text{type\_of} := \text{Mem\_}, \text{offset} := \text{max\_linear\_offset}\#))(\text{type\_of}(a'))$
{-8} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-9} Mem?(a′‘type_of)
{-10} $0 \leq a'‘\text{offset}$
{-11} $a'‘\text{offset} < \text{max\_linear\_offset}$
{-12} cons?[Byte](bl′)
{-13} is_linear_plain_memory?(pm′)
{-14} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-15} pm′‘states(s′)

{1} OK?((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))

Using lemma pm_linear_resolve_read_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1‘1, Read)(s!1)), length(t!1‘2)), union(pm_phy‘ro_addr, pm_phy‘rw_addr)),

we get 3 subgoals:

`split_linear_read_side_effects_ok.1.1.1.1.1.1.1:`

{-1} (address_block(data(linear_resolve(t'`1, Read)($s'$)), length($t'$`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t'`1, Read)(s'))
{-3} union(pm′`ro_addr, pm′`rw_addr)(t'`1)
{-4} cons?(t'`2)
{-5} $a' \le t'$`1
{-6} t'`1 + length($t'$`2) ≤ a' + length(bl′)
{-7} type_of(t'`1) = type_of(a')
{-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-9} t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10} rem(expt(2, min_page))(offset(t'`1)) + length($t'$`2) ≤ expt(2, min_page)
{-11} Mem?(a'`type_of)
{-12} 0 ≤ a'`offset
{-13} a'`offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a', length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-17} pm′`states(s')

{1} OK?((linear_resolve(t'`1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy`mem)(pa, $t'$`2, TRUE)(s')))
(s'))

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_transformers_ok_read_side_effects_block,

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.1`.

`split_linear_read_side_effects_ok.1.1.1.1.1.1.2:`

{-1}	is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2}	union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-3}	cons?(t′‘2)
{-4}	a′ ≤ t′‘1
{-5}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6}	type_of(t′‘1) = type_of(a′)
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10}	Mem?(a′‘type_of)
{-11}	0 ≤ a′‘offset
{-12}	a′‘offset < max_linear_offset
{-13}	cons?[Byte](bl′)
{-14}	is_linear_plain_memory?(pm′)
{-15}	(address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-16}	pm′‘states(s′)
{1}	(address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw
{2}	OK?((linear_resolve(t′‘1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′))) (s′))

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

`split_linear_read_side_effects_ok.1.1.1.1.1.1.2.1:`

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1) ∧
a′ ≤ t′‘1 ∧
t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧
rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
∧ (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
⊃
(address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-4} cons?(t′‘2)
{-5} a′ ≤ t′‘1
{-6} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7} type_of(t′‘1) = type_of(a′)
{-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-17} pm′‘states(s′)

{1} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.2.1`.
`split_linear_read_side_effects_ok.1.1.1.1.1.1.2.2:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} Mem?(a′‘type_of)
{-11} 0 ≤ a′‘offset
{-12} a′‘offset < max_linear_offset
{-13} cons?[Byte](bl′)
{-14} is_linear_plain_memory?(pm′)
{-15} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-16} pm′‘states(s′)

{1} length[Byte](t′‘2) > 0
{2} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.2.2`.

`split_linear_read_side_effects_ok.1.1.1.1.1.1.2.3:`

```
{-1}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'`1, Read)(s'))
{-2}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
{-3}  cons?(t'`2)
{-4}  a' ≤ t'`1
{-5}  t'`1 + length(t'`2) ≤ a' + length(bl')
{-6}  type_of(t'`1) = type_of(a')
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-8}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-9}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-10} Mem?(a'`type_of)
{-11} 0 ≤ a'`offset
{-12} a'`offset < max_linear_offset
{-13} cons?[Byte](bl')
{-14} is_linear_plain_memory?(pm')
{-15} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-16} pm'`states(s')
------------------------------------------------------------------------
 {1}  length[Byte](bl') > 0
 {2}  (address_block(data(linear_resolve(t'`1, Read)(s')), length(t'`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.2.3`.

`split_linear_read_side_effects_ok.1.1.1.1.1.1.3:`

```
{-1}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'`1, Read)(s'))
{-2}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
{-3}  cons?(t'`2)
{-4}  a' ≤ t'`1
{-5}  t'`1 + length(t'`2) ≤ a' + length(bl')
{-6}  type_of(t'`1) = type_of(a')
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-8}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-9}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-10} Mem?(a'`type_of)
{-11} 0 ≤ a'`offset
{-12} a'`offset < max_linear_offset
{-13} cons?[Byte](bl')
{-14} is_linear_plain_memory?(pm')
{-15} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-16} pm'`states(s')
------------------------------------------------------------------------
 {1}  OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t'`1, Read)(s'))
 {2}  OK?((linear_resolve(t'`1, Read) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_read_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s')))
           (s'))
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.1.3`.

`split_linear_read_side_effects_ok.1.1.1.1.1.2:`

{-1}	cons?(t'`2)
{-2}	$a' \leq t'$`1
{-3}	t'`1 + length($t'$`2) $\leq a'$ + length(bl$'$)
{-4}	type_of(t'`1) = type_of(a')
{-5}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t'$`1
{-6}	t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}	rem(expt(2, min_page))(offset(t'`1)) + length($t'$`2) $\leq$ expt(2, min_page)
{-8}	Mem?(a'`type_of)
{-9}	$0 \leq a'$`offset
{-10}	a'`offset < max_linear_offset
{-11}	cons?[Byte](bl$'$)
{-12}	is_linear_plain_memory?(pm$'$)
{-13}	(address_block(a', length(bl$'$)) $\subseteq$ (pm$'$`ro_addr $\cup$ pm$'$`rw_addr))
{-14}	pm$'$`states(s')
{1}	union(pm$'$`ro_addr, pm$'$`rw_addr)(t'`1)
{2}	OK?((linear_resolve(t'`1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s'))) (s'))

Hiding formulas: (-7 -12 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.1.2`.

`split_linear_read_side_effects_ok.1.1.1.1.2:`

```
{-1}  every(λ (t_1: [Address, list[Byte]]):
                cons?(t_1‘2) ∧
                 a′ ≤ t_1‘1 ∧
                  t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
                   type_of(t_1‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
                     t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
                       expt(2, min_page))
          (split(min_page, a′, bl′))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
                cons?(s‘2) ∧
                 a′ ≤ s‘1 ∧
                  s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
                   type_of(s‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
                     s‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                (cons?(t‘2) ∧
                   a′ ≤ t‘1 ∧
                    t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
                     type_of(t‘1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
                       t‘1 <
                        reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                ∧
                rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
                 expt(2, min_page))
          (split(min_page, a′, bl′))
{-3}  Mem?(a′‘type_of)
{-4}  0 ≤ a′‘offset
{-5}  a′‘offset < max_linear_offset
{-6}  cons?[Byte](bl′)
{-7}  is_linear_plain_memory?(pm′)
{-8}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-9}  pm′‘states(s′)
-------------------------------------------------------------------------
{1}   ∀ (s: [Memory_Address_4G, list[Byte]]):
        reg_base(Mem(0))(type_of(a′)) ≤ s‘1 ∧
         type_of(s‘1) = type_of(a′) ∧
          s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧ a′ ≤ s‘1 ∧ cons?(s‘2)
        ⊃ Mem?(Mem(max_linear_offset)‘type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_read_side_effect_in_page(e)(s′)))
          (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.2`.

`split_linear_read_side_effects_ok.1.1.1.1.3:`

```
{-1}  every(λ (t_1: [Address, list[Byte]]):
                cons?(t_1‘2) ∧
                 a′ ≤ t_1‘1 ∧
                  t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
                   type_of(t_1‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
                     t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
                cons?(s‘2) ∧
                 a′ ≤ s‘1 ∧
                  s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
                   type_of(s‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
                     s‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                (cons?(t‘2) ∧
                   a′ ≤ t‘1 ∧
                    t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
                     type_of(t‘1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
                       t‘1 <
                        reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                ∧
                rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
                 expt(2, min_page))
           (split(min_page, a′, bl′))
{-3}  Mem?(a′‘type_of)
{-4}  0 ≤ a′‘offset
{-5}  a′‘offset < max_linear_offset
{-6}  cons?[Byte](bl′)
{-7}  is_linear_plain_memory?(pm′)
{-8}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-9}  pm′‘states(s′)
-------------------------------------------------------------------------
{1}   ∀ (s: [Memory_Address_4G, list[Byte]]):
        type_of(s‘1) = type_of(a′) ∧
         s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧ a′ ≤ s‘1 ∧ cons?(s‘2)
         ⊃ Mem?(Mem(0)‘type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_read_side_effect_in_page(e)(s′)))
           (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.1.3`.

`split_linear_read_side_effects_ok.1.1.1.2:`

{-1} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-2} Mem?(a′‘type_of)
{-3} 0 ≤ a′‘offset
{-4} a′‘offset < max_linear_offset
{-5} cons?[Byte](bl′)
{-6} is_linear_plain_memory?(pm′)
{-7} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8} pm′‘states(s′)

{1} ∀ (t: [Address, list[Byte]]):
reg_base(Mem(0))(type_of(a′)) ≤ t‘1 ∧
type_of(t‘1) = type_of(a′) ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧ a′ ≤ t‘1 ∧ cons?(t‘2)
⊃ Mem?(Mem(max_linear_offset)‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.2`.

`split_linear_read_side_effects_ok.1.1.1.3:`

{-1} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-2} Mem?(a′‘type_of)
{-3} 0 ≤ a′‘offset
{-4} a′‘offset < max_linear_offset
{-5} cons?[Byte](bl′)
{-6} is_linear_plain_memory?(pm′)
{-7} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8} pm′‘states(s′)

{1} ∀ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧ a′ ≤ t‘1 ∧ cons?(t‘2)
⊃ Mem?(Mem(0)‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.1.3`.

split_linear_read_side_effects_ok.1.1.2:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
reg_base(Mem(0))(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(max_linear_offset)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.2`.

`split_linear_read_side_effects_ok.1.1.3:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(0)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.1.3`.

`split_linear_read_side_effects_ok.1.2:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                 a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                 cons?(t`2) ∧
                  a' ≤ t`1 ∧ t`1 + length(t`2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
                type_of(t`1) = type_of(a') ∧
                 reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t`1 ∧
                  t`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'`type_of)
{-7}  0 ≤ a'`offset
{-8}  a'`offset < max_linear_offset
{-9}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-13} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(Mem(0))(type_of(a')) ≤ t`1 ∧ type_of(t`1) = type_of(a') ⊃
         Mem?(Mem(max_linear_offset)`type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_read_side_effect_in_page(e)(s')))
           (split(min_page, a', bl'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.2`.

`split_linear_read_side_effects_ok.1.3:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t‘2) ∧
                 a' ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
              rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
              type_of(t‘1) = type_of(a') ∧
               reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
                t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
              a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'‘type_of)
{-7}  0 ≤ a'‘offset
{-8}  a'‘offset < max_linear_offset
{-9}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ (pm'‘ro_addr ∪ pm'‘rw_addr))
{-13} pm'‘states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        type_of(t‘1) = type_of(a') ⊃ Mem?(Mem(0)‘type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              OK?(linear_read_side_effect_in_page(e)(s')))
           (split(min_page, a', bl'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.1.3`.

`split_linear_read_side_effects_ok.2:`

{-1} every(λ (a₂: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3} every(λ (a₂: Address, bl2: list[Byte]):
a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-4} Mem?(a′`type_of)
{-5} 0 ≤ a′`offset
{-6} a′`offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-8} cons?[Byte](bl′)
{-9} is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-11} pm′`states(s′)

{1} a′ + length(bl′) ≤
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_ok.2`.

split_linear_read_side_effects_ok.3:

{-1} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-8} cons?[Byte](bl′)
{-9} is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-11} pm′‘states(s′)

{1} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ a′
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Expanding the definition of reg_base,
Expanding the definition of <=,
which is trivially true.
This completes the proof of split_linear_read_side_effects_ok.3.
Q.E.D.

C.117.79 Linear_Memory_Properties.split_linear_write_side_effects_ok

Terse proof for split_linear_write_side_effects_ok.

split_linear_write_side_effects_ok:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
a: Memory_Address_4G, bl: (cons?[Byte])):
is_linear_plain_memory?(pm) ∧
(address_block(a, length(bl)) ⊆ pm‘rw_addr) ∧ pm‘states(s)
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s)))
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: Mem!! max_linear!! min_linear!!
Using lemma split_range,
Using lemma split_no_null,
Using lemma split_type,
Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_write_side_effects_ok.1:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a', bl'))
{-2}  every(λ (t: [Address, list[Byte]]):
                type_of(t`1) = type_of(a') ∧
                 reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t`1 ∧
                  t`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
            (split(min_page, a', bl'))
{-3}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
            (split(min_page, a', bl'))
{-5}  Mem?(a'`type_of)
{-6}  0 ≤ a'`offset
{-7}  a'`offset < max_linear_offset
{-8}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl')
{-9}  cons?[Byte](bl')
{-10} is_linear_plain_memory?(pm')
{-11} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-12} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s')))
            (split(min_page, a', bl'))
```

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

`split_linear_write_side_effects_ok.1.1:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-14} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq$ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) $\leq$
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) $\leq$
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq t$‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) $\leq$ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2$ ∧ a_2 + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) $\leq$ expt(2, min_page))
(split(min_page, a', bl′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) $\leq t$‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
(split(min_page, a', bl′))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),
Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
we get 3 subgoals:
`split_linear_write_side_effects_ok.1.1.1.1:`

```
{-1}  every(λ (t_1: [Address, list[Byte]]):
                cons?(t_1`2) ∧
                 a′ ≤ t_1`1 ∧
                  t_1`1 + length(t_1`2) ≤ a′ + length(bl′) ∧
                   type_of(t_1`1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ t_1`1 ∧
                     t_1`1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(t_1`1)) + length(t_1`2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
                cons?(s`2) ∧
                 a′ ≤ s`1 ∧
                  s`1 + length(s`2) ≤ a′ + length(bl′) ∧
                   type_of(s`1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ s`1 ∧
                     s`1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(s`1)) + length(s`2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                (cons?(t`2) ∧
                   a′ ≤ t`1 ∧
                    t`1 + length(t`2) ≤ a′ + length(bl′) ∧
                     type_of(t`1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t`1 ∧
                       t`1 <
                        reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                 ∧
                 rem(expt(2, min_page))(offset(t`1)) + length(t`2) ≤
                  expt(2, min_page))
           (split(min_page, a′, bl′))
{-3}  Mem?(a′`type_of)
{-4}  0 ≤ a′`offset
{-5}  a′`offset < max_linear_offset
{-6}  cons?[Byte](bl′)
{-7}  is_linear_plain_memory?(pm′)
{-8}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-9}  pm′`states(s′)
------------------------------------------------------------
{1}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)))
           (split(min_page, a′, bl′))
```

Using lemma every_implied,
we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1.1.1:

{-1} (∀ (t: [Memory_Address_4G, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page)
⊃ OK?(linear_write_side_effect_in_page(t)(s′)))
∧
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} cons?[Byte](bl′)
{-8} is_linear_plain_memory?(pm′)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_write_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on pm!1`rw_addr(t!1`1),

we get 2 subgoals:

`split_linear_write_side_effects_ok.1.1.1.1.1.1:`

{-1} pm′`rw_addr(t′`1)
{-2} cons?(t′`2)
{-3} $a' \leq t'`1$
{-4} $t'`1 + \text{length}(t'`2) \leq a' + \text{length}(\text{bl}')$
{-5} type_of(t′`1) = type_of(a′)
{-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-7} $t'`1 <$ reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-8} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-9} Mem?(a′`type_of)
{-10} $0 \leq a'`\text{offset}$
{-11} $a'`\text{offset} <$ max_linear_offset
{-12} cons?[Byte](bl′)
{-13} is_linear_plain_memory?(pm′)
{-14} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-15} pm′`states(s′)

{1} OK?((linear_resolve(t′`1, Write) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_write_side_effect(pm_phy`mem)(pa, t′`2, TRUE)(s′)))
(s′))

Using lemma pm_linear_resolve_write_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1`1, Write)(s!1)), length(t!1`2)), pm_phy`rw_addr),

we get 3 subgoals:

split_linear_write_side_effects_ok.1.1.1.1.1.1.1:

{-1} (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-3} pm′‘rw_addr(t′‘1)
{-4} cons?(t′‘2)
{-5} a′ ≤ t′‘1
{-6} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7} type_of(t′‘1) = type_of(a′)
{-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-17} pm′‘states(s′)

{1} OK?((linear_resolve(t′‘1, Write) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_write_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))

Using lemma linear_resolve_states,
Using lemma pm_states,
Using lemma pm_plain_phy,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

split_linear_write_side_effects_ok.1.1.1.1.1.1.1:

{-1} is_linear_plain_memory?(pm′)
{-2} plain_memory?(pm_phy)
{-3} pm′‘states = pm_phy‘states
{-4} pm′‘states(s′)
{-5} OK?(linear_resolve(t′‘1, Write)(s′))
{-6} pm_phy‘states(state(linear_resolve(t′‘1, Write)(s′)))
{-7} (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-8} pm′‘rw_addr(t′‘1)
{-9} cons?(t′‘2)
{-10} a′ ≤ t′‘1
{-11} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-12} type_of(t′‘1) = type_of(a′)
{-13} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-14} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-15} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-16} Mem?(a′‘type_of)
{-17} 0 ≤ a′‘offset
{-18} a′‘offset < max_linear_offset
{-19} cons?[Byte](bl′)
{-20} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{1} OK?(memory_write_side_effect(pm_phy‘mem)
(data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))

Using lemma plain_memory_transformers_ok_write_side_effects_block,

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

Rewriting using subset_reflexive, matching in *,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.1.1`.

`split_linear_write_side_effects_ok.1.1.1.1.1.1.1.2:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′`states = pm_phy`states
{-4}	pm′`states(s′)
{-5}	OK?(linear_resolve(t′`1, Write)(s′))
{-6}	(address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
{-7}	pm′`rw_addr(t′`1)
{-8}	cons?(t′`2)
{-9}	a′ ≤ t′`1
{-10}	t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-11}	type_of(t′`1) = type_of(a′)
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-13}	t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-14}	rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-15}	Mem?(a′`type_of)
{-16}	0 ≤ a′`offset
{-17}	a′`offset < max_linear_offset
{-18}	cons?[Byte](bl′)
{-19}	(address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{1}	union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{2}	OK?(memory_write_side_effect(pm_phy`mem) (data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′))

Rewriting using union_right, matching in *,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.1.2`.

`split_linear_write_side_effects_ok.1.1.1.1.1.1.2:`

{-1}	is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2}	pm′‘rw_addr(t′‘1)
{-3}	cons?(t′‘2)
{-4}	a′ ≤ t′‘1
{-5}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6}	type_of(t′‘1) = type_of(a′)
{-7}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10}	Mem?(a′‘type_of)
{-11}	0 ≤ a′‘offset
{-12}	a′‘offset < max_linear_offset
{-13}	cons?[Byte](bl′)
{-14}	is_linear_plain_memory?(pm′)
{-15}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-16}	pm′‘states(s′)
{1}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{2}	OK?((linear_resolve(t′‘1, Write) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′))) (s′))

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

`split_linear_write_side_effects_ok.1.1.1.1.1.1.2.1:`

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘rw_addr(t′‘1) ∧
a′ ≤ t′‘1 ∧
t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧
rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
∧ (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
⊃
(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-3} pm′‘rw_addr(t′‘1)
{-4} cons?(t′‘2)
{-5} a′ ≤ t′‘1
{-6} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7} type_of(t′‘1) = type_of(a′)
{-8} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-17} pm′‘states(s′)

{1} (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.2.1`.
`split_linear_write_side_effects_ok.1.1.1.1.1.1.2.2:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2} pm′‘rw_addr(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} Mem?(a′‘type_of)
{-11} 0 ≤ a′‘offset
{-12} a′‘offset < max_linear_offset
{-13} cons?[Byte](bl′)
{-14} is_linear_plain_memory?(pm′)
{-15} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-16} pm′‘states(s′)

{1} length[Byte](t′‘2) > 0
{2} (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.2.2`.

`split_linear_write_side_effects_ok.1.1.1.1.1.1.2.3`:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2}  pm′‘rw_addr(t′‘1)
{-3}  cons?(t′‘2)
{-4}  a′ ≤ t′‘1
{-5}  t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6}  type_of(t′‘1) = type_of(a′)
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8}  t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}  rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} Mem?(a′‘type_of)
{-11} 0 ≤ a′‘offset
{-12} a′‘offset < max_linear_offset
{-13} cons?[Byte](bl′)
{-14} is_linear_plain_memory?(pm′)
{-15} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-16} pm′‘states(s′)
------------------------------------------------------------------------
 {1}  length[Byte](bl′) > 0
 {2}  (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.2.3`.

`split_linear_write_side_effects_ok.1.1.1.1.1.1.3`:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2}  pm′‘rw_addr(t′‘1)
{-3}  cons?(t′‘2)
{-4}  a′ ≤ t′‘1
{-5}  t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6}  type_of(t′‘1) = type_of(a′)
{-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8}  t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9}  rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} Mem?(a′‘type_of)
{-11} 0 ≤ a′‘offset
{-12} a′‘offset < max_linear_offset
{-13} cons?[Byte](bl′)
{-14} is_linear_plain_memory?(pm′)
{-15} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-16} pm′‘states(s′)
------------------------------------------------------------------------
 {1}  OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t′‘1, Write)(s′))
 {2}  OK?((linear_resolve(t′‘1, Write) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_write_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
           (s′))
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.1.3`.

`split_linear_write_side_effects_ok.1.1.1.1.1.2:`

```
{-1}   cons?(t'`2)
{-2}   a' <= t'`1
{-3}   t'`1 + length(t'`2) <= a' + length(bl')
{-4}   type_of(t'`1) = type_of(a')
{-5}   reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) <= t'`1
{-6}   t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}   rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) <= expt(2, min_page)
{-8}   Mem?(a'`type_of)
{-9}   0 <= a'`offset
{-10}  a'`offset < max_linear_offset
{-11}  cons?[Byte](bl')
{-12}  is_linear_plain_memory?(pm')
{-13}  (address_block(a', length(bl')) <= pm'`rw_addr)
{-14}  pm'`states(s')
-------------------------------------------------------------------------
 {1}   pm'`rw_addr(t'`1)
 {2}   OK?((linear_resolve(t'`1, Write) ##
              (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                 memory_write_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s')))
             (s'))
```

Hiding formulas: (-7 -12 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.1.2`.

`split_linear_write_side_effects_ok.1.1.1.1.2:`

```
{-1}  every(λ (t_1: [Address, list[Byte]]):
                cons?(t_1‘2) ∧
                 a′ ≤ t_1‘1 ∧
                  t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
                   type_of(t_1‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
                     t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
                       expt(2, min_page))
          (split(min_page, a′, bl′))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
                cons?(s‘2) ∧
                 a′ ≤ s‘1 ∧
                  s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
                   type_of(s‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
                     s‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                (cons?(t‘2) ∧
                   a′ ≤ t‘1 ∧
                    t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
                     type_of(t‘1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
                       t‘1 <
                        reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                ∧
                rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
                 expt(2, min_page))
          (split(min_page, a′, bl′))
{-3}  Mem?(a′‘type_of)
{-4}  0 ≤ a′‘offset
{-5}  a′‘offset < max_linear_offset
{-6}  cons?[Byte](bl′)
{-7}  is_linear_plain_memory?(pm′)
{-8}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9}  pm′‘states(s′)
-------------------------------------------------------------------------------
{1}   ∀ (s: [Memory_Address_4G, list[Byte]]):
        reg_base(Mem(0))(type_of(a′)) ≤ s‘1 ∧
         type_of(s‘1) = type_of(a′) ∧
          s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧ a′ ≤ s‘1 ∧ cons?(s‘2)
        ⊃ Mem?(Mem(max_linear_offset)‘type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)))
          (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.2`.

`split_linear_write_side_effects_ok.1.1.1.1.3:`

```
{-1}  every(λ (t_1: [Address, list[Byte]]):
                cons?(t_1`2) ∧
                 a′ ≤ t_1`1 ∧
                  t_1`1 + length(t_1`2) ≤ a′ + length(bl′) ∧
                   type_of(t_1`1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ t_1`1 ∧
                     t_1`1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(t_1`1)) + length(t_1`2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
                cons?(s`2) ∧
                 a′ ≤ s`1 ∧
                  s`1 + length(s`2) ≤ a′ + length(bl′) ∧
                   type_of(s`1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ s`1 ∧
                     s`1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(s`1)) + length(s`2) ≤
                       expt(2, min_page))
           (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                (cons?(t`2) ∧
                   a′ ≤ t`1 ∧
                    t`1 + length(t`2) ≤ a′ + length(bl′) ∧
                     type_of(t`1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t`1 ∧
                       t`1 <
                        reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                ∧
                rem(expt(2, min_page))(offset(t`1)) + length(t`2) ≤
                 expt(2, min_page))
           (split(min_page, a′, bl′))
{-3}  Mem?(a′`type_of)
{-4}  0 ≤ a′`offset
{-5}  a′`offset < max_linear_offset
{-6}  cons?[Byte](bl′)
{-7}  is_linear_plain_memory?(pm′)
{-8}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-9}  pm′`states(s′)
------------------------------------------------------------------------
{1}   ∀ (s: [Memory_Address_4G, list[Byte]]):
        type_of(s`1) = type_of(a′) ∧
         s`1 + length(s`2) ≤ a′ + length(bl′) ∧ a′ ≤ s`1 ∧ cons?(s`2)
         ⊃ Mem?(Mem(0)`type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)))
           (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.1.3`.

`split_linear_write_side_effects_ok.1.1.1.2:`

{-1} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-2} Mem?(a′‘type_of)
{-3} 0 ≤ a′‘offset
{-4} a′‘offset < max_linear_offset
{-5} cons?[Byte](bl′)
{-6} is_linear_plain_memory?(pm′)
{-7} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-8} pm′‘states(s′)

{1} ∀ (t: [Address, list[Byte]]):
reg_base(Mem(0))(type_of(a′)) ≤ t‘1 ∧
type_of(t‘1) = type_of(a′) ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧ a′ ≤ t‘1 ∧ cons?(t‘2)
⊃ Mem?(Mem(max_linear_offset)‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.2`.

`split_linear_write_side_effects_ok.1.1.1.3:`

```
{-1}  every(λ (t: [Address, list[Byte]]):
                (cons?(t`2) ∧
                   a′ ≤ t`1 ∧
                    t`1 + length(t`2) ≤ a′ + length(bl′) ∧
                     type_of(t`1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t`1 ∧
                       t`1 <
                         reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                 ∧
                 rem(expt(2, min_page))(offset(t`1)) + length(t`2) ≤
                  expt(2, min_page))
            (split(min_page, a′, bl′))
{-2}  Mem?(a′`type_of)
{-3}  0 ≤ a′`offset
{-4}  a′`offset < max_linear_offset
{-5}  cons?[Byte](bl′)
{-6}  is_linear_plain_memory?(pm′)
{-7}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-8}  pm′`states(s′)
-------------------------------------------------------------------------------
{1}   ∀ (t: [Address, list[Byte]]):
        type_of(t`1) = type_of(a′) ∧
         t`1 + length(t`2) ≤ a′ + length(bl′) ∧ a′ ≤ t`1 ∧ cons?(t`2)
         ⊃ Mem?(Mem(0)`type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)))
            (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.1.3`.

split_linear_write_side_effects_ok.1.1.2:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
reg_base(Mem(0))(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(max_linear_offset)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.2`.

`split_linear_write_side_effects_ok.1.1.3:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(0)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.1.3`.

`split_linear_write_side_effects_ok.1.2`:

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                 a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                 cons?(t`2) ∧
                  a' ≤ t`1 ∧ t`1 + length(t`2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
                 rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
                 type_of(t`1) = type_of(a') ∧
                  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t`1 ∧
                   t`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
            (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                 a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
            (split(min_page, a', bl'))
{-6}  Mem?(a'`type_of)
{-7}  0 ≤ a'`offset
{-8}  a'`offset < max_linear_offset
{-9}  every(λ (x: number):
                 number_field_pred(x) ∧
                  real_pred(x) ∧
                   rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-13} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(Mem(0))(type_of(a')) ≤ t`1 ∧ type_of(t`1) = type_of(a') ⊃
         Mem?(Mem(max_linear_offset)`type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                 OK?(linear_write_side_effect_in_page(e)(s')))
            (split(min_page, a', bl'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.2`.

`split_linear_write_side_effects_ok.1.3:`

{-1} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-2} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))
{-4} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-5} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-6} Mem?(a′‘type_of)
{-7} 0 ≤ a′‘offset
{-8} a′‘offset < max_linear_offset
{-9} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-10} cons?[Byte](bl′)
{-11} is_linear_plain_memory?(pm′)
{-12} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-13} pm′‘states(s′)

{1} ∀ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ⊃ Mem?(Mem(0)‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.1.3`.

`split_linear_write_side_effects_ok.2:`

{-1} every(λ (a₂: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3} every(λ (a₂: Address, bl2: list[Byte]):
a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-4} Mem?(a′`type_of)
{-5} 0 ≤ a′`offset
{-6} a′`offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-8} cons?[Byte](bl′)
{-9} is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-11} pm′`states(s′)

{1} a′ + length(bl′) ≤
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_ok.2`.

split_linear_write_side_effects_ok.3:

{-1} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-8} cons?[Byte](bl′)
{-9} is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-11} pm′‘states(s′)

{1} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ a′
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)))
(split(min_page, a′, bl′))

Expanding the definition of reg_base,
Expanding the definition of <=,
which is trivially true.
This completes the proof of split_linear_write_side_effects_ok.3.
Q.E.D.

C.117.80 Linear_Memory_Properties.split_linear_read_side_effects_data

Terse proof for split_linear_read_side_effects_data.

split_linear_read_side_effects_data:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
a: Memory_Address_4G, bl: (cons?[Byte])):
is_linear_plain_memory?(pm) ∧
(address_block(a, length(bl)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧ pm‘states(s)
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s)) ⊃
data(linear_read_side_effect_in_page(e)(s)) = e‘2)
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: Mem!! max_linear!! min_linear!!
Using lemma split_range,
Using lemma split_no_null,
Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_read_side_effects_data.1:`

{-1} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))
{-3} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-4} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-5} Mem?(a′‘type_of)
{-6} 0 ≤ a′‘offset
{-7} a′‘offset < max_linear_offset
{-8} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-9} cons?[Byte](bl′)
{-10} is_linear_plain_memory?(pm′)
{-11} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-12} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

`split_linear_read_side_effects_data.1.1:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-14} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_data.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2$ ∧ a_2 + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a', bl′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
(split(min_page, a', bl′))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),
Installing automatic rewrites from: (max_linear! Mem! ##! expr_2_super! expr_2_super_res!)
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
we get 2 subgoals:
`split_linear_read_side_effects_data.1.1.1.1:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a' ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a' + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a' ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a' + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))

{-2} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a' ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a' + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))

{-3} Mem?(a'‘type_of)

{-4} 0 ≤ a'‘offset

{-5} a'‘offset < max_linear_offset

{-6} cons?[Byte](bl′)

{-7} is_linear_plain_memory?(pm′)

{-8} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-9} pm′‘states(s')

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s')) ⊃
data(linear_read_side_effect_in_page(e)(s')) = e‘2)
(split(min_page, a', bl′))

Using lemma every_implied,
we get 2 subgoals:

split_linear_read_side_effects_data.1.1.1.1.1:

{-1} (∀ (t: [Memory_Address_4G, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page)
⊃
OK?(linear_read_side_effect_in_page(t)(s′)) ⊃
data(linear_read_side_effect_in_page(t)(s′)) = t‘2)
∧
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_read_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on union(pm!1‘ro_addr, pm!1‘rw_addr)(t!1‘1),

we get 2 subgoals:

`split_linear_read_side_effects_data.1.1.1.1.1.1:`

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-2} cons?(t′‘2)
{-3} $a' \leq t'‘1$
{-4} $t'‘1 + \text{length}(t'‘2) \leq a' + \text{length}(\text{bl}')$
{-5} type_of(t′‘1) = type_of(a′)
{-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-7} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-8} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-9} OK?((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
{-10} Mem?(a′‘type_of)
{-11} 0 ≤ a′‘offset
{-12} a′‘offset < max_linear_offset
{-13} cons?[Byte](bl′)
{-14} is_linear_plain_memory?(pm′)
{-15} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-16} pm′‘states(s′)

{1} data((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
= t′‘2

Using lemma pm_linear_resolve_read_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1‘1, Read)(s!1)), length(t!1‘2)), union(pm_phy‘ro_addr, pm_phy‘rw_addr)),

we get 3 subgoals:

`split_linear_read_side_effects_data.1.1.1.1.1.1.1:`

```
{-1}  (address_block(data(linear_resolve(t'‘1, Read)(s')), length(t'‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'‘1, Read)(s'))
{-3}  union(pm'‘ro_addr, pm'‘rw_addr)(t'‘1)
{-4}  cons?(t'‘2)
{-5}  a' ≤ t'‘1
{-6}  t'‘1 + length(t'‘2) ≤ a' + length(bl')
{-7}  type_of(t'‘1) = type_of(a')
{-8}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'‘1
{-9}  t'‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10} rem(expt(2, min_page))(offset(t'‘1)) + length(t'‘2) ≤ expt(2, min_page)
{-11} OK?((linear_resolve(t'‘1, Read) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_read_side_effect(pm_phy‘mem)(pa, t'‘2, TRUE)(s')))
           (s'))
{-12} Mem?(a'‘type_of)
{-13} 0 ≤ a'‘offset
{-14} a'‘offset < max_linear_offset
{-15} cons?[Byte](bl')
{-16} is_linear_plain_memory?(pm')
{-17} (address_block(a', length(bl')) ⊆ (pm'‘ro_addr ∪ pm'‘rw_addr))
{-18} pm'‘states(s')
------------------------------------------------------------------------
 {1}  data((linear_resolve(t'‘1, Read) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_read_side_effect(pm_phy‘mem)(pa, t'‘2, TRUE)(s')))
           (s'))
      = t'‘2
```

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-2 -3 -4 -5 -6 -8),

Expanding the definition of side_effect_content_unchanged,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.1`.

`split_linear_read_side_effects_data.1.1.1.1.1.1.2:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} OK?((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-17} pm′‘states(s′)

{1} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{2} data((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
= t′‘2

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

`split_linear_read_side_effects_data.1.1.1.1.1.1.2.1:`

{-1}	is_linear_plain_memory?(pm′) ∧ pm′‘states(s′) ∧ union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1) ∧ a′ ≤ t′‘1 ∧ t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧ rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page) ∧ (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr)) ⊃ (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘r
{-2}	is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-3}	union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-4}	cons?(t′‘2)
{-5}	a′ ≤ t′‘1
{-6}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7}	type_of(t′‘1) = type_of(a′)
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11}	OK?((linear_resolve(t′‘1, Read) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′))) (s′))
{-12}	Mem?(a′‘type_of)
{-13}	0 ≤ a′‘offset
{-14}	a′‘offset < max_linear_offset
{-15}	cons?[Byte](bl′)
{-16}	is_linear_plain_memory?(pm′)
{-17}	(address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-18}	pm′‘states(s′)
{1}	(address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.2.1`.

`split_linear_read_side_effects_data.1.1.1.1.1.1.2.2:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} OK?((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-17} pm′‘states(s′)

{1} length[Byte](t′‘2) > 0
{2} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.2.2`.
`split_linear_read_side_effects_data.1.1.1.1.1.1.2.3:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} OK?((linear_resolve(t′‘1, Read) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-17} pm′‘states(s′)

{1} length[Byte](bl′) > 0
{2} (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.2.3`.

`split_linear_read_side_effects_data.1.1.1.1.1.1.3`:

```
 {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
 {-2}  union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
 {-3}  cons?(t′‘2)
 {-4}  a′ ≤ t′‘1
 {-5}  t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
 {-6}  type_of(t′‘1) = type_of(a′)
 {-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
 {-8}  t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
 {-9}  rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10}  OK?((linear_resolve(t′‘1, Read) ##
              (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                 memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
            (s′))
{-11}  Mem?(a′‘type_of)
{-12}  0 ≤ a′‘offset
{-13}  a′‘offset < max_linear_offset
{-14}  cons?[Byte](bl′)
{-15}  is_linear_plain_memory?(pm′)
{-16}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-17}  pm′‘states(s′)
-------------------------------------------------------------------------------
 {1}   OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t′‘1, Read)(s′))
 {2}   data((linear_resolve(t′‘1, Read) ##
              (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                 memory_read_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
            (s′))
       = t′‘2
```

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.1.3`.

`split_linear_read_side_effects_data.1.1.1.1.1.2:`

```
{-1}  cons?(t'`2)
{-2}  a' ≤ t'`1
{-3}  t'`1 + length(t'`2) ≤ a' + length(bl')
{-4}  type_of(t'`1) = type_of(a')
{-5}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-6}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-8}  OK?((linear_resolve(t'`1, Read) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_read_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s')))
           (s'))
{-9}  Mem?(a'`type_of)
{-10} 0 ≤ a'`offset
{-11} a'`offset < max_linear_offset
{-12} cons?[Byte](bl')
{-13} is_linear_plain_memory?(pm')
{-14} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-15} pm'`states(s')
------------------------------------------------------------------------
 {1}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
 {2}  data((linear_resolve(t'`1, Read) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_read_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s')))
           (s'))
        = t'`2
```

Hiding formulas: (-7 -8 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.1.2`.

`split_linear_read_side_effects_data.1.1.1.1.2:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a' ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a' + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a' ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a' + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))

{-2} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a' ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a' + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))

{-3} Mem?(a'‘type_of)

{-4} 0 ≤ a'‘offset

{-5} a'‘offset < max_linear_offset

{-6} cons?[Byte](bl′)

{-7} is_linear_plain_memory?(pm′)

{-8} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-9} pm′‘states(s')

{1} ∀ (s: [Memory_Address_4G, list[Byte]]):
type_of(s‘1) = type_of(a') ∧
s‘1 + length(s‘2) ≤ a' + length(bl′) ∧ a' ≤ s‘1 ∧ cons?(s‘2)
⊃ Mem?(Mem(0)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s')) ⊃
data(linear_read_side_effect_in_page(e)(s')) = e‘2)
(split(min_page, a', bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.1.2`.

`split_linear_read_side_effects_data.1.1.1.2:`

```
{-1}  every(λ (t: [Address, list[Byte]]):
                (cons?(t`2) ∧
                   a′ ≤ t`1 ∧
                    t`1 + length(t`2) ≤ a′ + length(bl′) ∧
                     type_of(t`1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t`1 ∧
                       t`1 <
                         reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                 ∧
                 rem(expt(2, min_page))(offset(t`1)) + length(t`2) ≤
                  expt(2, min_page))
            (split(min_page, a′, bl′))
{-2}  Mem?(a′`type_of)
{-3}  0 ≤ a′`offset
{-4}  a′`offset < max_linear_offset
{-5}  cons?[Byte](bl′)
{-6}  is_linear_plain_memory?(pm′)
{-7}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-8}  pm′`states(s′)
------------------------------------------------------------------------
{1}   ∀ (t: [Address, list[Byte]]):
        type_of(t`1) = type_of(a′) ∧
         t`1 + length(t`2) ≤ a′ + length(bl′) ∧ a′ ≤ t`1 ∧ cons?(t`2)
         ⊃ Mem?(Mem(0)`type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
                 data(linear_read_side_effect_in_page(e)(s′)) = e`2)
            (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.1.2`.

`split_linear_read_side_effects_data.1.1.2:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
reg_base(Mem(0))(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(max_linear_offset)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.2`.

split_linear_read_side_effects_data.1.1.3:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(0)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.1.3`.

`split_linear_read_side_effects_data.1.2:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t`2) ∧
                 a' ≤ t`1 ∧ t`1 + length(t`2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
              rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
              type_of(t`1) = type_of(a') ∧
               reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t`1 ∧
                t`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
              a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'`type_of)
{-7}  0 ≤ a'`offset
{-8}  a'`offset < max_linear_offset
{-9}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-13} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(Mem(0))(type_of(a')) ≤ t`1 ∧ type_of(t`1) = type_of(a') ⊃
         Mem?(Mem(max_linear_offset)`type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              OK?(linear_read_side_effect_in_page(e)(s')) ⊃
               data(linear_read_side_effect_in_page(e)(s')) = e`2)
           (split(min_page, a', bl'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.2`.

`split_linear_read_side_effects_data.1.3:`

{-1} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-2} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))
{-4} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-5} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-6} Mem?(a′‘type_of)
{-7} 0 ≤ a′‘offset
{-8} a′‘offset < max_linear_offset
{-9} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-10} cons?[Byte](bl′)
{-11} is_linear_plain_memory?(pm′)
{-12} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-13} pm′‘states(s′)

{1} ∀ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ⊃ Mem?(Mem(0)‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.1.3`.

`split_linear_read_side_effects_data.2:`

```
  {-1}   every(λ (a₂: Address, bl2: list[Byte]):
                   rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
               (split(min_page, a′, bl′))
  {-2}   every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
  {-3}   every(λ (a₂: Address, bl2: list[Byte]):
                   a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
               (split(min_page, a′, bl′))
  {-4}   Mem?(a′`type_of)
  {-5}   0 ≤ a′`offset
  {-6}   a′`offset < max_linear_offset
  {-7}   every(λ (x: number):
                   number_field_pred(x) ∧
                    real_pred(x) ∧
                     rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
               (bl′)
  {-8}   cons?[Byte](bl′)
  {-9}   is_linear_plain_memory?(pm′)
  {-10}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
  {-11}  pm′`states(s′)
  ------------------------------------------------------------------------
  {1}    a′ + length(bl′) ≤
          reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {2}    every(λ (e: [Memory_Address_4G, list[Byte]]):
                   OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
                    data(linear_read_side_effect_in_page(e)(s′)) = e`2)
               (split(min_page, a′, bl′))
```

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_data.2`.

`split_linear_read_side_effects_data.3:`

{-1} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-8} cons?[Byte](bl′)
{-9} is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-11} pm′‘states(s′)

{1} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ a′
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_read_side_effect_in_page(e)(s′)) ⊃
data(linear_read_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Expanding the definition of reg_base,
Expanding the definition of <=,
which is trivially true.
This completes the proof of `split_linear_read_side_effects_data.3`.
Q.E.D.

C.117.81 Linear_Memory_Properties.split_linear_write_side_effects_data

Terse proof for `split_linear_write_side_effects_data`.

`split_linear_write_side_effects_data:`

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy],
a: Memory_Address_4G, bl: (cons?[Byte])):
is_linear_plain_memory?(pm) ∧
(address_block(a, length(bl)) ⊆ pm‘rw_addr) ∧ pm‘states(s)
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s)) ⊃
data(linear_write_side_effect_in_page(e)(s)) = e‘2)
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: Mem!! max_linear!! min_linear!!
Using lemma split_range,
Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_write_side_effects_data.1:`

{-1} every(λ (a2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))
{-2} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))
{-3} every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-4} every(λ (a2: Address, bl2: list[Byte]):
a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
{-5} Mem?(a′‘type_of)
{-6} 0 ≤ a′‘offset
{-7} a′‘offset < max_linear_offset
{-8} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-9} cons?[Byte](bl′)
{-10} is_linear_plain_memory?(pm′)
{-11} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-12} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

`split_linear_write_side_effects_data.1.1:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-14} pm′‘states(s′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2$ ∧ a_2 + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a', bl′))
{-5} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
(split(min_page, a', bl′))
{-6} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -3 -4 -5 -6 -7 -11),
Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
we get 3 subgoals:
`split_linear_write_side_effects_data.1.1.1.1:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a' ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a' + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a' ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a' + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))

{-2} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a' ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a' + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))

{-3} Mem?(a'‘type_of)

{-4} 0 ≤ a'‘offset

{-5} a'‘offset < max_linear_offset

{-6} cons?[Byte](bl′)

{-7} is_linear_plain_memory?(pm′)

{-8} (address_block(a', length(bl′)) ⊆ pm′‘rw_addr)

{-9} pm′‘states(s')

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s')) ⊃
data(linear_write_side_effect_in_page(e)(s')) = e‘2)
(split(min_page, a', bl′))

Using lemma every_implied,
we get 3 subgoals:

split_linear_write_side_effects_data.1.1.1.1.1:

{-1} (∀ (t: [Memory_Address_4G, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page)
⊃
OK?(linear_write_side_effect_in_page(t)(s′)) ⊃
data(linear_write_side_effect_in_page(t)(s′)) = t‘2)
∧
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 2),

Expanding the definition of linear_write_side_effect_in_page,

Repeatedly Skolemizing and flattening,

Case splitting on pm!1‘rw_addr(t!1‘1),

we get 2 subgoals:

`split_linear_write_side_effects_data.1.1.1.1.1.1:`

{-1} $\mathrm{pm}'\text{`}\mathrm{rw\_addr}(t'\text{`}1)$
{-2} $\mathrm{cons?}(t'\text{`}2)$
{-3} $a' \leq t'\text{`}1$
{-4} $t'\text{`}1 + \mathrm{length}(t'\text{`}2) \leq a' + \mathrm{length}(\mathrm{bl}')$
{-5} $\mathrm{type\_of}(t'\text{`}1) = \mathrm{type\_of}(a')$
{-6} $\mathrm{reg\_base}((\#\mathrm{type\_of} := \mathrm{Mem\_},\ \mathrm{offset} := 0\#))(\mathrm{type\_of}(a')) \leq t'\text{`}1$
{-7} $t'\text{`}1 < \mathrm{reg\_size}((\#\mathrm{type\_of} := \mathrm{Mem\_},\ \mathrm{offset} := \mathrm{max\_linear\_offset}\#))(\mathrm{type\_of}(a'))$
{-8} $\mathrm{rem}(\mathrm{expt}(2,\ \mathrm{min\_page}))(\mathrm{offset}(t'\text{`}1)) + \mathrm{length}(t'\text{`}2) \leq \mathrm{expt}(2,\ \mathrm{min\_page})$
{-9} $\mathrm{OK?}((\mathrm{linear\_resolve}(t'\text{`}1,\ \mathrm{Write})\ \#\#\ (\lambda\ (\mathrm{pa}\colon \mathrm{Memory\_Address\_4G})(\mathrm{ns}\colon \mathrm{Linear\_memory}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}])\colon \mathrm{memory\_write\_side\_effect}(\mathrm{pm\_phy}\text{`}\mathrm{mem})(\mathrm{pa},\ t'\text{`}2,\ \mathrm{TRUE})(s')))\ (s'))$
{-10} $\mathrm{Mem?}(a'\text{`}\mathrm{type\_of})$
{-11} $0 \leq a'\text{`}\mathrm{offset}$
{-12} $a'\text{`}\mathrm{offset} < \mathrm{max\_linear\_offset}$
{-13} $\mathrm{cons?}[\mathrm{Byte}](\mathrm{bl}')$
{-14} $\mathrm{is\_linear\_plain\_memory?}(\mathrm{pm}')$
{-15} $(\mathrm{address\_block}(a',\ \mathrm{length}(\mathrm{bl}')) \subseteq \mathrm{pm}'\text{`}\mathrm{rw\_addr})$
{-16} $\mathrm{pm}'\text{`}\mathrm{states}(s')$

{1} $\mathrm{data}((\mathrm{linear\_resolve}(t'\text{`}1,\ \mathrm{Write})\ \#\#\ (\lambda\ (\mathrm{pa}\colon \mathrm{Memory\_Address\_4G})(\mathrm{ns}\colon \mathrm{Linear\_memory}[\mathrm{Physical\_memory},\ \mathrm{pm\_phy}])\colon \mathrm{memory\_write\_side\_effect}(\mathrm{pm\_phy}\text{`}\mathrm{mem})(\mathrm{pa},\ t'\text{`}2,\ \mathrm{TRUE})(s')))\ (s'))$
$= t'\text{`}2$

Using lemma pm_linear_resolve_write_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1‘1, Write)(s!1)), length(t!1‘2)), pm_phy‘rw_addr),

we get 3 subgoals:

`split_linear_write_side_effects_data.1.1.1.1.1.1.1:`

```
{-1}  (address_block(data(linear_resolve(t'‘1, Write)(s')), length(t'‘2)) ⊆ pm_phy‘rw_addr)
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'‘1, Write)(s'))
{-3}  pm'‘rw_addr(t'‘1)
{-4}  cons?(t'‘2)
{-5}  a' ≤ t'‘1
{-6}  t'‘1 + length(t'‘2) ≤ a' + length(bl')
{-7}  type_of(t'‘1) = type_of(a')
{-8}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'‘1
{-9}  t'‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-10} rem(expt(2, min_page))(offset(t'‘1)) + length(t'‘2) ≤ expt(2, min_page)
{-11} OK?((linear_resolve(t'‘1, Write) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_write_side_effect(pm_phy‘mem)(pa, t'‘2, TRUE)(s')))
           (s'))
{-12} Mem?(a'‘type_of)
{-13} 0 ≤ a'‘offset
{-14} a'‘offset < max_linear_offset
{-15} cons?[Byte](bl')
{-16} is_linear_plain_memory?(pm')
{-17} (address_block(a', length(bl')) ⊆ pm'‘rw_addr)
{-18} pm'‘states(s')
-------------------------------------------------------------------------------
 {1}  data((linear_resolve(t'‘1, Write) ##
             (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                memory_write_side_effect(pm_phy‘mem)(pa, t'‘2, TRUE)(s')))
           (s'))
      = t'‘2
```

Using lemma linear_resolve_states,

Using lemma pm_states,

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`split_linear_write_side_effects_data.1.1.1.1.1.1.1.1:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′`states = pm_phy`states
{-4}	pm′`states(s′)
{-5}	OK?(linear_resolve(t′`1, Write)(s′))
{-6}	pm_phy`states(state(linear_resolve(t′`1, Write)(s′)))
{-7}	(address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
{-8}	pm′`rw_addr(t′`1)
{-9}	cons?(t′`2)
{-10}	a′ ≤ t′`1
{-11}	t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-12}	type_of(t′`1) = type_of(a′)
{-13}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-14}	t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-15}	rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-16}	OK?(memory_write_side_effect(pm_phy`mem) (data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′))
{-17}	Mem?(a′`type_of)
{-18}	0 ≤ a′`offset
{-19}	a′`offset < max_linear_offset
{-20}	cons?[Byte](bl′)
{-21}	(address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{1}	data(memory_write_side_effect(pm_phy`mem) (data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′)) = t′`2

Expanding the definition of plain_memory?,

Applying disjunctive simplification to flatten sequent,

Hiding formulas: (-2 -3 -4 -5 -6 -7),

Expanding the definition of side_effect_content_unchanged,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.1.1`.

`split_linear_write_side_effects_data.1.1.1.1.1.1.1.2:`

{-1}	is_linear_plain_memory?(pm′)
{-2}	plain_memory?(pm_phy)
{-3}	pm′‘states = pm_phy‘states
{-4}	pm′‘states(s′)
{-5}	OK?(linear_resolve(t′‘1, Write)(s′))
{-6}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-7}	pm′‘rw_addr(t′‘1)
{-8}	cons?(t′‘2)
{-9}	a′ ≤ t′‘1
{-10}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-11}	type_of(t′‘1) = type_of(a′)
{-12}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-13}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-14}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-15}	OK?(memory_write_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))
{-16}	Mem?(a′‘type_of)
{-17}	0 ≤ a′‘offset
{-18}	a′‘offset < max_linear_offset
{-19}	cons?[Byte](bl′)
{-20}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{1}	union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{2}	data(memory_write_side_effect(pm_phy‘mem) (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′)) = t′‘2

Keeping (-7 1) and hiding *,

Rewriting using union_right, matching in *,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.1.2`.

`split_linear_write_side_effects_data.1.1.1.1.1.1.2:`

```
  {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
  {-2}  pm′`rw_addr(t′`1)
  {-3}  cons?(t′`2)
  {-4}  a′ ≤ t′`1
  {-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
  {-6}  type_of(t′`1) = type_of(a′)
  {-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
  {-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
  {-10} OK?((linear_resolve(t′`1, Write) ##
               (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                  memory_write_side_effect(pm_phy`mem)(pa, t′`2, TRUE)(s′)))
             (s′))
  {-11} Mem?(a′`type_of)
  {-12} 0 ≤ a′`offset
  {-13} a′`offset < max_linear_offset
  {-14} cons?[Byte](bl′)
  {-15} is_linear_plain_memory?(pm′)
  {-16} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
  {-17} pm′`states(s′)
  ------------------------------------------------------------------------
  {1}   (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
  {2}   data((linear_resolve(t′`1, Write) ##
               (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                  memory_write_side_effect(pm_phy`mem)(pa, t′`2, TRUE)(s′)))
             (s′))
        = t′`2
```

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

`split_linear_write_side_effects_data.1.1.1.1.1.1.2.1:`

{-1}	is_linear_plain_memory?(pm′) ∧ pm′‘states(s′) ∧ pm′‘rw_addr(t′‘1) ∧ a′ ≤ t′‘1 ∧ t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧ rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page) ∧ (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr) ⊃ (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
{-2}	is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-3}	pm′‘rw_addr(t′‘1)
{-4}	cons?(t′‘2)
{-5}	a′ ≤ t′‘1
{-6}	t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-7}	type_of(t′‘1) = type_of(a′)
{-8}	reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-9}	t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-10}	rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-11}	OK?((linear_resolve(t′‘1, Write) ## (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]): memory_write_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′))) (s′))
{-12}	Mem?(a′‘type_of)
{-13}	0 ≤ a′‘offset
{-14}	a′‘offset < max_linear_offset
{-15}	cons?[Byte](bl′)
{-16}	is_linear_plain_memory?(pm′)
{-17}	(address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-18}	pm′‘states(s′)
{1}	(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.2.1`.

`split_linear_write_side_effects_data.1.1.1.1.1.1.2.2:`

```
  {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
  {-2}  pm′`rw_addr(t′`1)
  {-3}  cons?(t′`2)
  {-4}  a′ ≤ t′`1
  {-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
  {-6}  type_of(t′`1) = type_of(a′)
  {-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
  {-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
  {-10} OK?((linear_resolve(t′`1, Write) ##
               (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                  memory_write_side_effect(pm_phy`mem)(pa, t′`2, TRUE)(s′)))
             (s′))
  {-11} Mem?(a′`type_of)
  {-12} 0 ≤ a′`offset
  {-13} a′`offset < max_linear_offset
  {-14} cons?[Byte](bl′)
  {-15} is_linear_plain_memory?(pm′)
  {-16} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
  {-17} pm′`states(s′)
  ------------------------------------------------------------
  {1}   length[Byte](t′`2) > 0
  {2}   (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
```

Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.2.2`.
`split_linear_write_side_effects_data.1.1.1.1.1.1.2.3:`

```
  {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
  {-2}  pm′`rw_addr(t′`1)
  {-3}  cons?(t′`2)
  {-4}  a′ ≤ t′`1
  {-5}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
  {-6}  type_of(t′`1) = type_of(a′)
  {-7}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
  {-8}  t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {-9}  rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
  {-10} OK?((linear_resolve(t′`1, Write) ##
               (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                  memory_write_side_effect(pm_phy`mem)(pa, t′`2, TRUE)(s′)))
             (s′))
  {-11} Mem?(a′`type_of)
  {-12} 0 ≤ a′`offset
  {-13} a′`offset < max_linear_offset
  {-14} cons?[Byte](bl′)
  {-15} is_linear_plain_memory?(pm′)
  {-16} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
  {-17} pm′`states(s′)
  ------------------------------------------------------------
  {1}   length[Byte](bl′) > 0
  {2}   (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.2.3`.

`split_linear_write_side_effects_data.1.1.1.1.1.1.3:`

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2} pm′‘rw_addr(t′‘1)
{-3} cons?(t′‘2)
{-4} a′ ≤ t′‘1
{-5} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-6} type_of(t′‘1) = type_of(a′)
{-7} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-8} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-9} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-10} OK?((linear_resolve(t′‘1, Write) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_write_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)
{-15} is_linear_plain_memory?(pm′)
{-16} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-17} pm′‘states(s′)

{1} OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](t′‘1, Write)(s′))
{2} data((linear_resolve(t′‘1, Write) ##
(λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
memory_write_side_effect(pm_phy‘mem)(pa, t′‘2, TRUE)(s′)))
(s′))
= t′‘2

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.1.3`.

`split_linear_write_side_effects_data.1.1.1.1.1.2:`

```
  {-1}  cons?(t'`2)
  {-2}  a' <= t'`1
  {-3}  t'`1 + length(t'`2) <= a' + length(bl')
  {-4}  type_of(t'`1) = type_of(a')
  {-5}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) <= t'`1
  {-6}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
  {-7}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) <= expt(2, min_page)
  {-8}  OK?((linear_resolve(t'`1, Write) ##
               (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                  memory_write_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s')))
             (s'))
  {-9}  Mem?(a'`type_of)
  {-10} 0 <= a'`offset
  {-11} a'`offset < max_linear_offset
  {-12} cons?[Byte](bl')
  {-13} is_linear_plain_memory?(pm')
  {-14} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
  {-15} pm'`states(s')
  ------------------------------------------------------------------------
  {1}   pm'`rw_addr(t'`1)
  {2}   data((linear_resolve(t'`1, Write) ##
               (λ (pa: Memory_Address_4G)(ns: Linear_memory[Physical_memory, pm_phy]):
                  memory_write_side_effect(pm_phy`mem)(pa, t'`2, TRUE)(s')))
             (s'))
        = t'`2
```

Hiding formulas: (-7 -8 -13 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.1.2`.

`split_linear_write_side_effects_data.1.1.1.1.2:`

```
{-1}  every(λ (t_1: [Address, list[Byte]]):
                cons?(t_1‘2) ∧
                 a′ ≤ t_1‘1 ∧
                  t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
                   type_of(t_1‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
                     t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
                       expt(2, min_page))
          (split(min_page, a′, bl′))
      ⊃
      every(λ (s: [Memory_Address_4G, list[Byte]]):
                cons?(s‘2) ∧
                 a′ ≤ s‘1 ∧
                  s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
                   type_of(s‘1) = type_of(a′) ∧
                    reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
                     s‘1 < reg_size(max_linear)(type_of(a′)) ∧
                      rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
                       expt(2, min_page))
          (split(min_page, a′, bl′))
{-2}  every(λ (t: [Address, list[Byte]]):
                (cons?(t‘2) ∧
                   a′ ≤ t‘1 ∧
                    t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
                     type_of(t‘1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
                       t‘1 <
                        reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                ∧
                rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
                 expt(2, min_page))
          (split(min_page, a′, bl′))
{-3}  Mem?(a′‘type_of)
{-4}  0 ≤ a′‘offset
{-5}  a′‘offset < max_linear_offset
{-6}  cons?[Byte](bl′)
{-7}  is_linear_plain_memory?(pm′)
{-8}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-9}  pm′‘states(s′)
-------------------------------------------------------------------------
{1}   ∀ (s: [Memory_Address_4G, list[Byte]]):
        reg_base(Mem(0))(type_of(a′)) ≤ s‘1 ∧
         type_of(s‘1) = type_of(a′) ∧
          s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧ a′ ≤ s‘1 ∧ cons?(s‘2)
        ⊃ Mem?(Mem(max_linear_offset)‘type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
                 data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
          (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.2`.

`split_linear_write_side_effects_data.1.1.1.1.3:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a' ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a' + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a' ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a' + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
a' ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a' + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
t‘1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-3} Mem?(a'‘type_of)
{-4} 0 ≤ a'‘offset
{-5} a'‘offset < max_linear_offset
{-6} cons?[Byte](bl′)
{-7} is_linear_plain_memory?(pm′)
{-8} (address_block(a', length(bl′)) ⊆ pm′‘rw_addr)
{-9} pm′‘states(s')

{1} ∀ (s: [Memory_Address_4G, list[Byte]]):
type_of(s‘1) = type_of(a') ∧
s‘1 + length(s‘2) ≤ a' + length(bl′) ∧ a' ≤ s‘1 ∧ cons?(s‘2)
⊃ Mem?(Mem(0)‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s')) ⊃
data(linear_write_side_effect_in_page(e)(s')) = e‘2)
(split(min_page, a', bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.1.3`.

`split_linear_write_side_effects_data.1.1.1.2:`

```
{-1}  every(λ (t: [Address, list[Byte]]):
                (cons?(t‘2) ∧
                   a′ ≤ t‘1 ∧
                    t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
                     type_of(t‘1) = type_of(a′) ∧
                      reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
                       t‘1 <
                         reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
                 ∧
                 rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
                  expt(2, min_page))
            (split(min_page, a′, bl′))
{-2}  Mem?(a′‘type_of)
{-3}  0 ≤ a′‘offset
{-4}  a′‘offset < max_linear_offset
{-5}  cons?[Byte](bl′)
{-6}  is_linear_plain_memory?(pm′)
{-7}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-8}  pm′‘states(s′)
------------------------------------------------------------------------
{1}   ∀ (t: [Address, list[Byte]]):
        reg_base(Mem(0))(type_of(a′)) ≤ t‘1 ∧
         type_of(t‘1) = type_of(a′) ∧
          t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧ a′ ≤ t‘1 ∧ cons?(t‘2)
        ⊃ Mem?(Mem(max_linear_offset)‘type_of)
{2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
                 data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
            (split(min_page, a′, bl′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.2`.

`split_linear_write_side_effects_data.1.1.1.3:`

{-1} every(λ (t: [Address, list[Byte]]):
(cons?(t`2) ∧
a′ ≤ t`1 ∧
t`1 + length(t`2) ≤ a′ + length(bl′) ∧
type_of(t`1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t`1 ∧
t`1 <
reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
∧
rem(expt(2, min_page))(offset(t`1)) + length(t`2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-2} Mem?(a′`type_of)
{-3} 0 ≤ a′`offset
{-4} a′`offset < max_linear_offset
{-5} cons?[Byte](bl′)
{-6} is_linear_plain_memory?(pm′)
{-7} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-8} pm′`states(s′)

{1} ∀ (t: [Address, list[Byte]]):
type_of(t`1) = type_of(a′) ∧
t`1 + length(t`2) ≤ a′ + length(bl′) ∧ a′ ≤ t`1 ∧ cons?(t`2)
⊃ Mem?(Mem(0)`type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
data(linear_write_side_effect_in_page(e)(s′)) = e`2)
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.1.3`.

split_linear_write_side_effects_data.1.1.2:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
reg_base(Mem(0))(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(max_linear_offset)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.2`.

split_linear_write_side_effects_data.1.1.3:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-4} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
(split(min_page, a′, bl′))

{-5} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-7} Mem?(a′‘type_of)

{-8} 0 ≤ a′‘offset

{-9} a′‘offset < max_linear_offset

{-10} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-11} cons?[Byte](bl′)

{-12} is_linear_plain_memory?(pm′)

{-13} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-14} pm′‘states(s′)

{1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(Mem(0)‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
(split(min_page, a′, bl′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.1.3`.

`split_linear_write_side_effects_data.1.2`:

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t`2) ∧
                 a' ≤ t`1 ∧ t`1 + length(t`2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
              rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
              type_of(t`1) = type_of(a') ∧
               reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t`1 ∧
                t`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
              a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'`type_of)
{-7}  0 ≤ a'`offset
{-8}  a'`offset < max_linear_offset
{-9}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-13} pm'`states(s')
-------------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(Mem(0))(type_of(a')) ≤ t`1 ∧ type_of(t`1) = type_of(a') ⊃
         Mem?(Mem(max_linear_offset)`type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              OK?(linear_write_side_effect_in_page(e)(s')) ⊃
               data(linear_write_side_effect_in_page(e)(s')) = e`2)
           (split(min_page, a', bl'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.2`.

`split_linear_write_side_effects_data.1.3:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t‘2) ∧
                 a' ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
           (split(min_page, a', bl'))
{-3}  every(λ (t: [Address, list[Byte]]):
                type_of(t‘1) = type_of(a') ∧
                 reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t‘1 ∧
                  t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a')))
           (split(min_page, a', bl'))
{-4}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
           (split(min_page, a', bl'))
{-6}  Mem?(a'‘type_of)
{-7}  0 ≤ a'‘offset
{-8}  a'‘offset < max_linear_offset
{-9}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
           (bl')
{-10} cons?[Byte](bl')
{-11} is_linear_plain_memory?(pm')
{-12} (address_block(a', length(bl')) ⊆ pm'‘rw_addr)
{-13} pm'‘states(s')
------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        type_of(t‘1) = type_of(a') ⊃ Mem?(Mem(0)‘type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s')) ⊃
                 data(linear_write_side_effect_in_page(e)(s')) = e‘2)
           (split(min_page, a', bl'))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.1.3`.

`split_linear_write_side_effects_data.2:`

```
  {-1}   every(λ (a₂: Address, bl2: list[Byte]):
                   rem(expt(2, min_page))(offset(a₂)) + length(bl2) ≤ expt(2, min_page))
               (split(min_page, a′, bl′))
  {-2}   every(λ (a₂: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
  {-3}   every(λ (a₂: Address, bl2: list[Byte]):
                   a′ ≤ a₂ ∧ a₂ + length(bl2) ≤ a′ + length(bl′))
               (split(min_page, a′, bl′))
  {-4}   Mem?(a′`type_of)
  {-5}   0 ≤ a′`offset
  {-6}   a′`offset < max_linear_offset
  {-7}   every(λ (x: number):
                   number_field_pred(x) ∧
                    real_pred(x) ∧
                     rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
               (bl′)
  {-8}   cons?[Byte](bl′)
  {-9}   is_linear_plain_memory?(pm′)
  {-10}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
  {-11}  pm′`states(s′)
  ------------------------------------------------------------------------
  {1}    a′ + length(bl′) ≤
          reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {2}    every(λ (e: [Memory_Address_4G, list[Byte]]):
                   OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
                    data(linear_write_side_effect_in_page(e)(s′)) = e`2)
               (split(min_page, a′, bl′))
```

Hiding formulas: (-1 -2 -3 -7 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -6,

Expanding the definition of subset?,

Installing automatic rewrites from: (bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_data.2`.

`split_linear_write_side_effects_data.3:`

```
{-1}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a′, bl′))
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
            (split(min_page, a′, bl′))
{-4}  Mem?(a′‘type_of)
{-5}  0 ≤ a′‘offset
{-6}  a′‘offset < max_linear_offset
{-7}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-8}  cons?[Byte](bl′)
{-9}  is_linear_plain_memory?(pm′)
{-10} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-11} pm′‘states(s′)
------------------------------------------------------------------------
 {1}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ a′
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                OK?(linear_write_side_effect_in_page(e)(s′)) ⊃
                 data(linear_write_side_effect_in_page(e)(s′)) = e‘2)
            (split(min_page, a′, bl′))
```

Expanding the definition of reg_base,
Expanding the definition of <=,
which is trivially true.
This completes the proof of `split_linear_write_side_effects_data.3`.
Q.E.D.

C.117.82 Linear_Memory_Properties.split_linear_read_side_effects_unchanged_memory

Terse proof for `split_linear_read_side_effects_unchanged_memory`.

`split_linear_read_side_effects_unchanged_memory:`

```
------------------------------------------------------------------------
{1}  ∀ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])):
       is_linear_plain_memory?(pm) ∧
        (address_block(a, length(bl)) ⊆ (pm‘ro_addr ∪ pm‘rw_addr))
        ⊃
        every(λ (e: [Memory_Address_4G, list[Byte]]):
                  unchanged_memory_invariant?(pm_phy‘mem, pm‘states,
                                              singleton(expr_2_super(linear_read_side_effect_in_pa
                                                                     (e))),
                                              (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
              (split(min_page, a, bl))
```

Repeatedly Skolemizing and flattening,
Using lemma split_range,
Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Using lemma split_linear_read_side_effects_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_read_side_effects_unchanged_memory.1:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
            transformer_invariant?(pm'`states,
                                   singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
          (split(min_page, a', bl'))
{-4}  every(λ (a_2: Address, bl2: list[Byte]):
            rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
          (split(min_page, a', bl'))
{-5}  every(λ (t: [Address, list[Byte]]):
            type_of(t`1) = type_of(a') ∧
              reg_base(min_linear)(type_of(a')) ≤ t`1 ∧
                t`1 < reg_size(max_linear)(type_of(a')))
          (split(min_page, a', bl'))
{-6}  every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-7}  every(λ (a_2: Address, bl2: list[Byte]):
            a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
          (split(min_page, a', bl'))
{-8}  Mem?(a'`type_of)
{-9}  0 ≤ a'`offset
{-10} a'`offset < max_linear_offset
{-11} every(λ (x: number):
            number_field_pred(x) ∧
              real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
          (bl')
{-12} cons?[Byte](bl')
------------------------------------------------------------------------
 {1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
            unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                        singleton(expr_2_super(linear_read_side_effect_in_page
                                                               (e))),
                                        (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
          (split(min_page, a', bl'))
```

Using lemma every_conjunct_left,

Using lemma every_conjunct_left,

we get 3 subgoals:

`split_linear_read_side_effects_unchanged_memory.1.1:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} is_linear_plain_memory?(pm′)

{-4} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))
(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-7} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-8} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-9} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-10} Mem?(a′‘type_of)

{-11} 0 ≤ a′‘offset

{-12} a′‘offset < max_linear_offset

{-13} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-14} cons?[Byte](bl′)

2252 {1} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_pag
(e))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
(split(min_page, a′, bl′))

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2 \wedge a_2$ + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} is_linear_plain_memory?(pm′)
{-5} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
2254 {-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))
(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a', bl′))
{-8} every(λ (t: [Address, list[Byte]]):

Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],

we get 3 subgoals:

`split_linear_read_side_effects_unchanged_memory.1.1.1.1:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
$a' \leq s$‘1 ∧
s‘1 + length(s‘2) $\leq a'$ + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-3} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧

Using lemma every_conjunct_left,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1:

{-1} every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
∧
every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Memory_Address_4G, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
∧
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page
(t)))))
(split(min_page, a′, bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a′, bl′))
{-3} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -4 -5 -6 -7 -8 -9 -10 -16),

Installing automatic rewrites from: (max_linear! Mem! ##! expr_2_super! expr_2_super_res!)

Using lemma every_implied,

we get 2 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1:

{-1} (∀ (t: [Memory_Address_4G, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page)
∧
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page
(t))))
⊃
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(t))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
∧
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
a′ ≤ s‘1 ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧
type_of(s‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page)
∧
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_p
(s)))))
(split(min_page, a′, bl′))
⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_pag
(e))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
(split(min_page, a′, bl′))
{-2} every(λ (t: [Memory_Address_4G, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
∧
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(t)))
(split(min_page, a′, bl′))
{-3} is_linear_plain_memory?(pm′)
{-4} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5} Mem?(a′‘type_of)
{-6} 0 ≤ a′‘offset
{-7} a′‘offset < max_linear_offset
{-8} cons?[Byte](bl′)

{1} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page
(e))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 2),
Expanding the definition of linear_read_side_effect_in_page,
Repeatedly Skolemizing and flattening,
Case splitting on union(pm!1`ro_addr, pm!1`rw_addr)(t!1`1),
we get 2 subgoals:
`split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1:`

```
{-1}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
{-2}  cons?(t'`2)
{-3}  a' <= t'`1
{-4}  t'`1 + length(t'`2) <= a' + length(bl')
{-5}  type_of(t'`1) = type_of(a')
{-6}  reg_base(min_linear)(type_of(a')) <= t'`1
{-7}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-8}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) <= expt(2, min_page)
{-9}  transformer_invariant?(pm'`states,
                             singleton(expr_2_super(λ (s:
                                                        Linear_memory[Physical_memory, pm_phy]):
                                                      (linear_resolve(t'`1, Read) ##
                                                         (λ (pa: Memory_Address_4G)
                                                              (ns:
                                                                 Linear_memory
                                                                   [Physical_memory, pm_phy]):
                                                            memory_read_side_effect(pm_phy`mem)
                                                              (pa, t'`2, TRUE)(s)))
                                                        (s))))
{-10} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-11} Mem?(a'`type_of)
{-12} 0 <= a'`offset
{-13} a'`offset < max_linear_offset
{-14} cons?[Byte](bl')
------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                  singleton(expr_2_super(λ (s:
                                                             Linear_memory
                                                               [Physical_memory, pm_phy]):
                                                           (linear_resolve(t'`1, Read) ##
                                                              (λ
                                                                 (pa: Memory_Address_4G)
                                                                 (ns:
                                                                    Linear_memory
                                                                      [Physical_memory, pm_phy]):
                                                                 memory_read_side_effect(pm_phy`mem)
                                                                   (pa, t'`2, TRUE)(s)))
                                                             (s))),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Using lemma pm_states,

Using lemma pm_plain_phy,

Expanding the definition of unchanged_memory_invariant?,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -13,

Hiding formulas: -13,

Using lemma pm_linear_resolve_read_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1‘1, Read)(s!1)), length(t!1‘2)), union(pm_phy‘ro_addr, pm_phy‘rw_addr)),

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1:

```
{-1}  (address_block(data(linear_resolve(t'1, Read)(s')), length(t'2)) ⊆ (pm_phy'ro_addr ∪ pm_phy'rw_addr))
{-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Read)(s'))
{-3}  plain_memory?(pm_phy)
{-4}  pm''states = pm_phy'states
{-5}  union(pm''ro_addr, pm''rw_addr)(t'1)
{-6}  cons?(t'2)
{-7}  a' ≤ t'1
{-8}  t'1 + length(t'2) ≤ a' + length(bl')
{-9}  type_of(t'1) = type_of(a')
{-10} reg_base(min_linear)(type_of(a')) ≤ t'1
{-11} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-12} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
{-13} transformer_invariant?(pm''states,
                             {y |
                                y =
                                 expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                CASES linear_resolve(t'1, Read)(s) OF
                                                  OK(state, data):
                                                    memory_read_side_effect(pm_phy'mem)
                                                          (data, t'2, TRUE)(s),
                                                  Exception(ex_type, state):
                                                    Exception(ex_type, state),
                                                  Fatal: Fatal,
                                                  Hang: Hang
                                                ENDCASES)})
{-14} pm''states(s')
{-15} union(pm_phy'ro_addr, pm_phy'rw_addr)(a'')
{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         CASES linear_resolve(t'1, Read)(s) OF
                           OK(state, data):
                             memory_read_side_effect(pm_phy'mem)(data, t'2, TRUE)(s),
                           Exception(ex_type, state): Exception(ex_type, state),
                           Fatal: Fatal,
                           Hang: Hang
                         ENDCASES)
                       (s'))
{-17} OK?(memory_read(pm_phy'mem)(a'')(s'))
{-18} OK?(memory_read(pm_phy'mem)
                     (a'')
                     (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                           CASES linear_resolve(t'1, Read)(s) OF
                                             OK(state, data):
                                               memory_read_side_effect(pm_phy'mem)
                                                     (data, t'2, TRUE)(s),
                                             Exception(ex_type, state): Exception(ex_type, state),
                                             Fatal: Fatal,
                                             Hang: Hang
                                           ENDCASES)
                                         (s'))))
{-19} (address_block(a', length(bl')) ⊆ (pm''ro_addr ∪ pm''rw_addr))
{-20} Mem?(a'type_of)
{-21} 0 ≤ a'offset
{-22} a'offset < max_linear_offset                                                    2263
{-23} cons?[Byte](bl')
-------------------------------------------------------------------------------------------
 {1}  data(memory_read(pm_phy'mem)
                      (a'')
                      (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                            CASES linear_resolve(t'1, Read)(s) OF
                                              OK(state, data):
                                                memory_read_side_effect(pm_phy'mem)
```

Using lemma linear_resolve_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_invariant,

Using lemma unchanged_memory_invariant_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on memory_read_side_effect_super_transformers(pm_phy‘mem, union (pm_phy‘ro_addr, pm_phy‘rw_addr)) (expr_2_super (memory_read_side_effect (pm_phy‘mem) (data (linear_resolve (t!1‘1, Read)(s!1)), t!1‘2, TRUE))),

we get 2 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.1:

```
{-1}  memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr))
                                                  (expr_2_super(memory_read_side_effect(pm_phy`mem)
                                                                (data
                                                                  (linear_resolve
                                                                    (t'`1, Read)(s')),
                                                                  t'`2,
                                                                  TRUE)))
{-2}  unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
                                  ((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-3}  union(pm_phy`ro_addr, pm_phy`rw_addr)(a'')
{-4}  OK?(memory_read(pm_phy`mem)(a'')(s'))
{-5}  plain_memory?(pm_phy)
{-6}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
{-7}  pm'`states(s')
{-8}  OK?(linear_resolve(t'`1, Read)(s'))
{-9}  pm_phy`states(state(linear_resolve(t'`1, Read)(s')))
{-10} (address_block(data(linear_resolve(t'`1, Read)(s')), length(t'`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-11} pm'`states = pm_phy`states
{-12} cons?(t'`2)
{-13} a' ≤ t'`1
{-14} t'`1 + length(t'`2) ≤ a' + length(bl')
{-15} type_of(t'`1) = type_of(a')
{-16} reg_base(min_linear)(type_of(a')) ≤ t'`1
{-17} t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-18} rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-19} transformer_invariant?(pm'`states,
                             {y |
                               y =
                                expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                               CASES lin-
      ear_resolve(t'`1, Read)(s) OF
                                                 OK(state, data):
                                                   mem-
      ory_read_side_effect(pm_phy`mem)
                                                         (data, t'`2, TRUE)(s),
                                                 Exception(ex_type, state):
                                                   Exception(ex_type, state),
                                                 Fatal: Fatal,
                                                 Hang: Hang
                                                 ENDCASES)})
{-20} OK?(memory_read_side_effect(pm_phy`mem)
                  (data(linear_resolve(t'`1, Read)(s')), t'`2, TRUE)(s'))
{-21} OK?(OK(state(memory_read_side_effect(pm_phy`mem)
                           (data(linear_resolve(t'`1, Read)(s')), t'`2, TRUE)(s'))))
{-22} OK?(memory_read(pm_phy`mem)
                  (a'')
                  (state(memory_read_side_effect(pm_phy`mem)
                                    (data(linear_resolve(t'`1, Read)(s')), t'`2, TRUE)(s'))))
{-23} (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-24} Mem?(a'`type_of)
{-25} 0 ≤ a'`offset
{-26} a'`offset < max_linear_offset
{-27} cons?[Byte](bl')
-------------------------------------------------------------------------------
{1}   union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_add
            (memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)) ∪
          (expr_2_super(memory_read_side_effect(pm_phy`mem)
                                (data(linear_resolve(t'`1, Read)(s')), t'`2, TRUE)))
{2}   data(memory_read(pm_phy`mem)
                  (a'')
                  (state(memory_read_side_effect(pm_phy`mem)
                                    (data(linear_resolve(t'`1, Read)(s')), t'`2, TRUE)(s'))))
        = data(memory_read(pm_phy`mem)(a'')(s'))
```

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.1`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.1.2:

{-1} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
{-3} OK?(memory_read(pm_phy`mem)(a″)(s′))
{-4} plain_memory?(pm_phy)
{-5} union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-6} pm′`states(s′)
{-7} OK?(linear_resolve(t′`1, Read)(s′))
{-8} pm_phy`states(state(linear_resolve(t′`1, Read)(s′)))
{-9} (address_block(data(linear_resolve(t′`1, Read)(s′)), length(t′`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-10} pm′`states = pm_phy`states
{-11} cons?(t′`2)
{-12} a′ ≤ t′`1
{-13} t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-14} type_of(t′`1) = type_of(a′)
{-15} reg_base(min_linear)(type_of(a′)) ≤ t′`1
{-16} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-17} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-18} transformer_invariant?(pm′`states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES lin-
ear_resolve(t′`1, Read)(s) OF
OK(state, data):
mem-
ory_read_side_effect(pm_phy`mem)
(data, t′`2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-19} OK?(memory_read_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)(s′))
{-20} OK?(OK(state(memory_read_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)(s′))))
{-21} OK?(memory_read(pm_phy`mem)
(a″)
(state(memory_read_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)(s′))))
{-22} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-23} Mem?(a′`type_of)
{-24} 0 ≤ a′`offset
{-25} a′`offset < max_linear_offset
{-26} cons?[Byte](bl′)

{1} memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr))
(expr_2_super(memory_read_side_effect(pm_phy`mem)
(data
(linear_resolve
(t′`1, Read)(s′)),
t′`2,
TRUE)))
{2} union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_add
(memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)) ∪
(expr_2_super(memory_read_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)))
{3} data(memory_read(pm_phy`mem)
(a″)
(state(memory_read_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Read)(s′)), t′`2, TRUE)(s′))))
= data(memory_read(pm_phy`mem)(a″)(s′))

Hiding formulas: (-1 -11 2 3),

Expanding the definition of memory_read_side_effect_super_transformers,

Instantiating quantified variables,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2:

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Read)(s′))
{-2} plain_memory?(pm_phy)
{-3} pm′`states = pm_phy`states
{-4} union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-5} cons?(t′`2)
{-6} a′ ≤ t′`1
{-7} t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-8} type_of(t′`1) = type_of(a′)
{-9} reg_base(min_linear)(type_of(a′)) ≤ t′`1
{-10} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-11} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm′`states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′`1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy`mem)
(data, t′`2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-13} pm′`states(s′)
{-14} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′`1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy`mem)(data, t′`2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))
{-16} OK?(memory_read(pm_phy`mem)(a″)(s′))
{-17} OK?(memory_read(pm_phy`mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′`1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy`mem)
(data, t′`2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))))
{-18} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-19} Mem?(a′`type_of)
{-20} 0 ≤ a′`offset
{-21} a′`offset < max_linear_offset
{-22} cons?[Byte](bl′)

2269

{1} (address_block(data(linear_resolve(t′`1, Read)(s′)), length(t′`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
{2} data(memory_read(pm_phy`mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′`1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy`mem)

Hiding formulas: 2,

Using lemma same_page_address_block_read,

we get 3 subgoals:

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2.1:

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1) ∧
a′ ≤ t′‘1 ∧
t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧
rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
∧ (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
⊃
(address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-3} plain_memory?(pm_phy)
{-4} pm′‘states = pm_phy‘states
{-5} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-6} cons?(t′‘2)
{-7} a′ ≤ t′‘1
{-8} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-9} type_of(t′‘1) = type_of(a′)
{-10} reg_base(min_linear)(type_of(a′)) ≤ t′‘1
{-11} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-12} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-13} transformer_invariant?(pm′‘states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-14} pm′‘states(s′)
{-15} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(a″)
{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy‘mem)(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))
{-17} OK?(memory_read(pm_phy‘mem)(a″)(s′))
{-18} OK?(memory_read(pm_phy‘mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))))
{-19} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-20} Mem?(a′‘type_of)
{-21} 0 ≤ a′‘offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2.1`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2.2:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Read)(s′))
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  union(pm′`ro_addr, pm′`rw_addr)(t′`1)
{-5}  cons?(t′`2)
{-6}  a′ ≤ t′`1
{-7}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-8}  type_of(t′`1) = type_of(a′)
{-9}  reg_base(min_linear)(type_of(a′)) ≤ t′`1
{-10} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-11} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm′`states,
                             {y |
                                y =
                                 expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                CASES linear_resolve(t′`1, Read)(s) OF
                                                  OK(state, data):
                                                    memory_read_side_effect(pm_phy`mem)
                                                      (data, t′`2, TRUE)(s),
                                                  Exception(ex_type, state):
                                                    Exception(ex_type, state),
                                                  Fatal: Fatal,
                                                  Hang: Hang
                                                ENDCASES)})
{-13} pm′`states(s′)
{-14} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         CASES linear_resolve(t′`1, Read)(s) OF
                           OK(state, data):
                             memory_read_side_effect(pm_phy`mem)(data, t′`2, TRUE)(s),
                           Exception(ex_type, state): Exception(ex_type, state),
                           Fatal: Fatal,
                           Hang: Hang
                         ENDCASES)
                       (s′))
{-16} OK?(memory_read(pm_phy`mem)(a″)(s′))
{-17} OK?(memory_read(pm_phy`mem)
                     (a″)
                     (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                           CASES linear_resolve(t′`1, Read)(s) OF
                                             OK(state, data):
                                               memory_read_side_effect(pm_phy`mem)
                                                 (data, t′`2, TRUE)(s),
                                             Exception(ex_type, state): Exception(ex_type, state),
                                             Fatal: Fatal,
                                             Hang: Hang
                                           ENDCASES)
                                         (s′))))
{-18} (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-19} Mem?(a′`type_of)
{-20} 0 ≤ a′`offset
{-21} a′`offset < max_linear_offset
{-22} cons?[Byte](bl′)                                                    2273
-----------------------------------------------------------------------------
 {1}  length[Byte](t′`2) > 0
 {2}  (address_block(data(linear_resolve(t′`1, Read)(s′)), length(t′`2)) ⊆ (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2.2`.

```
split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.2.3:
```

```
  {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
  {-2}  plain_memory?(pm_phy)
  {-3}  pm′‘states = pm_phy‘states
  {-4}  union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
  {-5}  cons?(t′‘2)
  {-6}  a′ ≤ t′‘1
  {-7}  t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
  {-8}  type_of(t′‘1) = type_of(a′)
  {-9}  reg_base(min_linear)(type_of(a′)) ≤ t′‘1
 {-10}  t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
 {-11}  rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
 {-12}  transformer_invariant?(pm′‘states,
                               {y |
                                  y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                  CASES lin-
        ear_resolve(t′‘1, Read)(s) OF
                                                    OK(state, data):
                                                      mem-
        ory_read_side_effect(pm_phy‘mem)
                                                            (data, t′‘2, TRUE)(s),
                                                    Exception(ex_type, state):
                                                      Exception(ex_type, state),
                                                    Fatal: Fatal,
                                                    Hang: Hang
                                                    ENDCASES)})
 {-13}  pm′‘states(s′)
 {-14}  union(pm_phy‘ro_addr, pm_phy‘rw_addr)(a″)
 {-15}  OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                           CASES linear_resolve(t′‘1, Read)(s) OF
                             OK(state, data):
                               memory_read_side_effect(pm_phy‘mem)(data, t′‘2, TRUE)(s),
                             Exception(ex_type, state): Exception(ex_type, state),
                             Fatal: Fatal,
                             Hang: Hang
                             ENDCASES)
                         (s′))
 {-16}  OK?(memory_read(pm_phy‘mem)(a″)(s′))
 {-17}  OK?(memory_read(pm_phy‘mem)
                       (a″)
                       (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                             CASES linear_resolve(t′‘1, Read)(s) OF
                                               OK(state, data):
                                                 memory_read_side_effect(pm_phy‘mem)
                                                                        (data, t′‘2, TRUE)(s),
                                               Exception(ex_type, state): Excep-
        tion(ex_type, state),
                                               Fatal: Fatal,
                                               Hang: Hang
                                               ENDCASES)
                                           (s′))))
 {-18}  (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
 {-19}  Mem?(a′‘type_of)
 {-20}  0 ≤ a′‘offset
 {-21}  a′‘offset < max_linear_offset
 {-22}  cons?[Byte](bl′)                                                    2275
-------------------------------------------------------------------------------
  {1}   length[Byte](bl′) > 0
  {2}   (address_block(data(linear_resolve(t′‘1, Read)(s′)), length(t′‘2)) ⊆ (pm_phy‘ro_addr ∪ pm_phy‘rw_addr))
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.2.3`.

split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.1.3:

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Read)(s′))
{-2} plain_memory?(pm_phy)
{-3} pm′‘states = pm_phy‘states
{-4} union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
{-5} cons?(t′‘2)
{-6} a′ ≤ t′‘1
{-7} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-8} type_of(t′‘1) = type_of(a′)
{-9} reg_base(min_linear)(type_of(a′)) ≤ t′‘1
{-10} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-11} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm′‘states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-13} pm′‘states(s′)
{-14} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(a″)
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy‘mem)(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))
{-16} OK?(memory_read(pm_phy‘mem)(a″)(s′))
{-17} OK?(memory_read(pm_phy‘mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):
memory_read_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))))
{-18} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-19} Mem?(a′‘type_of)
{-20} 0 ≤ a′‘offset
{-21} a′‘offset < max_linear_offset
{-22} cons?[Byte](bl′)

2277

{1} OK?[Linear_memory[Physical_memory, pm_phy], Address]
(linear_resolve[Physical_memory, pm_phy](t′‘1, Read)(s′))
{2} data(memory_read(pm_phy‘mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Read)(s) OF
OK(state, data):

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.1.3`.

`split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.2`:

```
{-1}  cons?(t'`2)
{-2}  a' ≤ t'`1
{-3}  t'`1 + length(t'`2) ≤ a' + length(bl')
{-4}  type_of(t'`1) = type_of(a')
{-5}  reg_base(min_linear)(type_of(a')) ≤ t'`1
{-6}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-8}  transformer_invariant?(pm'`states,
                             singleton(expr_2_super(λ (s:
                                                      Lin-
      ear_memory[Physical_memory, pm_phy]):
                                                   (linear_resolve(t'`1, Read) ##
                                                      (λ (pa: Mem-
      ory_Address_4G)
                                                         (ns:
                                                            Linear_memory
                                                              [Physical_memory, pm_
                                                         mem-
      ory_read_side_effect(pm_phy`mem)
                                                            (pa, t'`2, TRUE)(s)))
                                                      (s))))
{-9}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-10} Mem?(a'`type_of)
{-11} 0 ≤ a'`offset
{-12} a'`offset < max_linear_offset
{-13} cons?[Byte](bl')
-------------------------------------------------------------------------------
 {1}  union(pm'`ro_addr, pm'`rw_addr)(t'`1)
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                  singleton(expr_2_super(λ (s:
                                                            Linear_memory
                                                              [Physical_memory, pm_
                                                         (linear_resolve(t'`1, Read) #
                                                            (λ
                                                               (pa: Mem-
      ory_Address_4G)
                                                               (ns:
                                                                  Linear_memory
                                                                    [Physical_memory, pr
                                                               mem-
      ory_read_side_effect(pm_phy`mem)
                                                               (pa, t'`2, TRUE)(s)))
                                                            (s))),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Hiding formulas: (-8 -7 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.2`.

`split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.2:`

{-1} every(λ (t: [Memory_Address_4G, list[Byte]]):
(cons?(t‘2) ∧
a′ ≤ t‘1 ∧
t‘1 + length(t‘2) ≤ a′ + length(bl′) ∧
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)) ∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
∧
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(t)))))
(split(min_page, a′, bl′))
{-2} is_linear_plain_memory?(pm′)
{-3} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} Mem?(a′‘type_of)
{-5} 0 ≤ a′‘offset
{-6} a′‘offset < max_linear_offset
{-7} cons?[Byte](bl′)

{1} ∀ (s: [Memory_Address_4G, list[Byte]]):
type_of(s‘1) = type_of(a′) ∧
s‘1 + length(s‘2) ≤ a′ + length(bl′) ∧ a′ ≤ s‘1 ∧ cons?(s‘2)
⊃ Mem?(min_linear‘type_of)
{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page
(e))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
(split(min_page, a′, bl′))

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.1.2`.

`split_linear_read_side_effects_unchanged_memory.1.1.1.1.2:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
$a' \leq s$‘1 ∧
s‘1 + length(s‘2) $\leq a'$ + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-3} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.2`.

`split_linear_read_side_effects_unchanged_memory.1.1.1.1.3:`

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(t_1‘1)) + length(t_1‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (s: [Memory_Address_4G, list[Byte]]):
cons?(s‘2) ∧
$a' \leq s$‘1 ∧
s‘1 + length(s‘2) $\leq a'$ + length(bl′) ∧
type_of(s‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ s‘1 ∧
s‘1 < reg_size(max_linear)(type_of(a')) ∧
rem(expt(2, min_page))(offset(s‘1)) + length(s‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-3} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.1.3`.

split_linear_read_side_effects_unchanged_memory.1.1.1.2:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \leq t$‘1 ∧
t‘1 + length(t‘2) $\leq a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \leq$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\leq a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \leq a_2$ ∧ a_2 + length(bl2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \leq t$‘1 ∧ t‘1 + length(t‘2) $\leq a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} is_linear_plain_memory?(pm′)
{-5} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
2284 {-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e))))
(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a', bl′))
{-8} every(λ (t: [Address, list[Byte]]):

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.2`.

split_linear_read_side_effects_unchanged_memory.1.1.1.3:

{-1} every(λ (t_1: [Address, list[Byte]]):
cons?(t_1‘2) ∧
$a' \le$ t_1‘1 ∧
t_1‘1 + length(t_1‘2) $\le a'$ + length(bl′) ∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
∧
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
(cons?(t‘2) ∧
$a' \le t$‘1 ∧
t‘1 + length(t‘2) $\le a'$ + length(bl′) ∧
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
∧
rem(expt(2, min_page))(offset(t‘1)) + length(t‘2) ≤
expt(2, min_page))
(split(min_page, a', bl′))
{-2} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \le t$‘1 ∧ t‘1 + length(t‘2) $\le a'$ + length(bl′))
(split(min_page, a', bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
$a' \le$ t_1‘1 ∧ t_1‘1 + length(t_1‘2) $\le a'$ + length(bl′))
∧
type_of(t_1‘1) = type_of(a') ∧
reg_base(min_linear)(type_of(a')) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a')))
(split(min_page, a', bl′))
{-3} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
$a' \le a_2 \wedge a_2$ + length(bl2) $\le a'$ + length(bl′))
(split(min_page, a', bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
$a' \le t$‘1 ∧ t‘1 + length(t‘2) $\le a'$ + length(bl′))
(split(min_page, a', bl′))
{-4} is_linear_plain_memory?(pm′)
{-5} (address_block(a', length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e))))
(split(min_page, a', bl′))
{-7} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a', bl′))
{-8} every(λ (t: [Address, list[Byte]]):

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.1.3`.

split_linear_read_side_effects_unchanged_memory.1.1.2:

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} is_linear_plain_memory?(pm′)

{-4} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))
(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-7} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-8} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-9} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-10} Mem?(a′‘type_of)

{-11} 0 ≤ a′‘offset

{-12} a′‘offset < max_linear_offset

{-13} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-14} cons?[Byte](bl′)

2288 {1} ∀ (t_1: [Address, list[Byte]]):
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(max_linear‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,

Expanding the definition of max_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.2`.

`split_linear_read_side_effects_unchanged_memory.1.1.3:`

{-1} every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
∧
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))
⊃
every(λ (t_1: [Address, list[Byte]]):
(cons?(t_1‘2) ∧
a′ ≤ t_1‘1 ∧ t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′))
∧
type_of(t_1‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t_1‘1 ∧
t_1‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-2} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′)) ∧
every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))
⊃
every(λ (t: [Address, list[Byte]]):
cons?(t‘2) ∧
a′ ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-3} is_linear_plain_memory?(pm′)

{-4} (address_block(a′, length(bl′)) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{-5} every(λ (e: [Memory_Address_4G, list[Byte]]):
transformer_invariant?(pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_page(e)))
(split(min_page, a′, bl′))

{-6} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
(split(min_page, a′, bl′))

{-7} every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a′) ∧
reg_base(min_linear)(type_of(a′)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a′)))
(split(min_page, a′, bl′))

{-8} every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))

{-9} every(λ (a_2: Address, bl2: list[Byte]):
a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
(split(min_page, a′, bl′))

{-10} Mem?(a′‘type_of)

{-11} 0 ≤ a′‘offset

{-12} a′‘offset < max_linear_offset

{-13} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)

{-14} cons?[Byte](bl′)

2290 {1} ∀ (t_1: [Address, list[Byte]]):
type_of(t_1‘1) = type_of(a′) ∧
t_1‘1 + length(t_1‘2) ≤ a′ + length(bl′) ∧
a′ ≤ t_1‘1 ∧ cons?(t_1‘2)
⊃ Mem?(min_linear‘type_of)

{2} every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(linear_read_side_effect_in_pag

Expanding the definition of min_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.1.3`.
`split_linear_read_side_effects_unchanged_memory.1.2`:

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                cons?(t`2) ∧
                 a' ≤ t`1 ∧ t`1 + length(t`2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  is_linear_plain_memory?(pm')
{-3}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
{-4}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm'`states,
                                       singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
             (split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
             (split(min_page, a', bl'))
{-6}  every(λ (t: [Address, list[Byte]]):
                type_of(t`1) = type_of(a') ∧
                 reg_base(min_linear)(type_of(a')) ≤ t`1 ∧
                  t`1 < reg_size(max_linear)(type_of(a')))
             (split(min_page, a', bl'))
{-7}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-8}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-9}  Mem?(a'`type_of)
{-10} 0 ≤ a'`offset
{-11} a'`offset < max_linear_offset
{-12} every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
             (bl')
{-13} cons?[Byte](bl')
------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        reg_base(min_linear)(type_of(a')) ≤ t`1 ∧ type_of(t`1) = type_of(a') ⊃
         Mem?(max_linear`type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                            singleton(expr_2_super(linear_read_side_effect_in_page
                                                                   (e))),
                                            (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
             (split(min_page, a', bl'))
```

Expanding the definition of max_linear,
Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.2`.

`split_linear_read_side_effects_unchanged_memory.1.3`:

```
{-1}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl')) ∧
       every(λ (a2: Address, bl2: list[Byte]):
                 a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
       ⊃
       every(λ (t: [Address, list[Byte]]):
                 cons?(t‘2) ∧
                  a' ≤ t‘1 ∧ t‘1 + length(t‘2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-2}  is_linear_plain_memory?(pm')
{-3}  (address_block(a', length(bl')) ⊆ (pm'‘ro_addr ∪ pm'‘rw_addr))
{-4}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                 transformer_invariant?(pm'‘states,
                                        singleton(expr_2_super(linear_read_side_effect_in_page(e))))
             (split(min_page, a', bl'))
{-5}  every(λ (a2: Address, bl2: list[Byte]):
                 rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
             (split(min_page, a', bl'))
{-6}  every(λ (t: [Address, list[Byte]]):
                 type_of(t‘1) = type_of(a') ∧
                  reg_base(min_linear)(type_of(a')) ≤ t‘1 ∧
                   t‘1 < reg_size(max_linear)(type_of(a')))
             (split(min_page, a', bl'))
{-7}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-8}  every(λ (a2: Address, bl2: list[Byte]):
                 a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
{-9}  Mem?(a'‘type_of)
{-10} 0 ≤ a'‘offset
{-11} a'‘offset < max_linear_offset
{-12} every(λ (x: number):
                 number_field_pred(x) ∧
                  real_pred(x) ∧
                   rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
             (bl')
{-13} cons?[Byte](bl')
-------------------------------------------------------------------------
 {1}  ∀ (t: [Address, list[Byte]]):
        type_of(t‘1) = type_of(a') ⊃ Mem?(min_linear‘type_of)
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                 unchanged_memory_invariant?(pm_phy‘mem, pm'‘states,
                                             singleton(expr_2_super(linear_read_side_effect_in_pag
                                                                    (e))),
                                             (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
             (split(min_page, a', bl'))
```

Expanding the definition of min_linear,

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.1.3`.

`split_linear_read_side_effects_unchanged_memory.2:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  (address_block(a′, length(bl′)) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
  {-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′`states,
                                       singleton(expr_2_super(linear_read_side_effect_in_page(e)))))
              (split(min_page, a′, bl′))
  {-4}  every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
              (split(min_page, a′, bl′))
  {-5}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
  {-6}  every(λ (a2: Address, bl2: list[Byte]):
                a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
              (split(min_page, a′, bl′))
  {-7}  Mem?(a′`type_of)
  {-8}  0 ≤ a′`offset
  {-9}  a′`offset < max_linear_offset
 {-10}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
              (bl′)
 {-11}  cons?[Byte](bl′)
--------------------------------------------------------------------------------
  {1}   a′ + length(bl′) ≤ reg_size(max_linear)(type_of(a′))
  {2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy`mem, pm′`states,
                                            singleton(expr_2_super(linear_read_side_effect_in_page
                                                                   (e))),
                                            (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
              (split(min_page, a′, bl′))
```

Hiding formulas: (-3 -4 -5 -6 -10 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_read_side_effects_unchanged_memory.2`.

`split_linear_read_side_effects_unchanged_memory.3:`

```
 {-1}  is_linear_plain_memory?(pm')
 {-2}  (address_block(a', length(bl')) ⊆ (pm'`ro_addr ∪ pm'`rw_addr))
 {-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                 transformer_invariant?(pm'`states,
                                        singleton(expr_2_super(linear_read_side_effect_in_page(e)))
             (split(min_page, a', bl'))
 {-4}  every(λ (a_2: Address, bl2: list[Byte]):
                 rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
             (split(min_page, a', bl'))
 {-5}  every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
 {-6}  every(λ (a_2: Address, bl2: list[Byte]):
                 a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
             (split(min_page, a', bl'))
 {-7}  Mem?(a'`type_of)
 {-8}  0 ≤ a'`offset
 {-9}  a'`offset < max_linear_offset
{-10}  every(λ (x: number):
                 number_field_pred(x) ∧
                  real_pred(x) ∧
                    rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
             (bl')
{-11}  cons?[Byte](bl')
-------------------------------------------------------------------------
  {1}  reg_base(min_linear)(type_of(a')) ≤ a'
  {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                 unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                             singleton(expr_2_super(linear_read_side_effect_in_pag
                                                                    (e))),
                                             (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
             (split(min_page, a', bl'))
```

Expanding the definition of reg_base,

Expanding the definition of min_linear,

Expanding the definition of Mem,

Expanding the definition of <=,

which is trivially true.

This completes the proof of `split_linear_read_side_effects_unchanged_memory.3`.

Q.E.D.

C.117.83 Linear_Memory_Properties.split_linear_write_side_effects_unchanged_memory

Terse proof for `split_linear_write_side_effects_unchanged_memory`.

split_linear_write_side_effects_unchanged_memory:

{1} ∀ (pm: Plain_Memory[Linear_memory], a: Memory_Address_4G, bl: (cons?[Byte])):
is_linear_plain_memory?(pm) ∧ (address_block(a, length(bl)) ⊆ pm‘rw_addr) ⊃
every(λ (e: [Memory_Address_4G, list[Byte]]):
unchanged_memory_invariant?(pm_phy‘mem, pm‘states,
singleton(expr_2_super(linear_write_side_effect_in_page(e))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (Mem! max_linear! min_linear!)

Using lemma split_range,

Using lemma split_no_null,

Using lemma split_type,

Using lemma split_pair_cross_size,

Using lemma split_linear_write_side_effects_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_linear_write_side_effects_unchanged_memory.1:`

```
{-1}  is_linear_plain_memory?(pm′)
{-2}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm′‘states,
                                     singleton(expr_2_super(linear_write_side_effect_in_page(e))
            (split(min_page, a′, bl′))
{-4}  every(λ (a2: Address, bl2: list[Byte]):
              rem(expt(2, min_page))(offset(a2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a′, bl′))
{-5}  every(λ (t: [Address, list[Byte]]):
              type_of(t‘1) = type_of(a′) ∧
               reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t‘1 ∧
                t‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′)))
            (split(min_page, a′, bl′))
{-6}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
{-7}  every(λ (a2: Address, bl2: list[Byte]):
              a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + length(bl′))
            (split(min_page, a′, bl′))
{-8}  Mem?(a′‘type_of)
{-9}  0 ≤ a′‘offset
{-10} a′‘offset < max_linear_offset
{-11} every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-12} cons?[Byte](bl′)
------------------------------------------------------------------------
 {1}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
                                          singleton(expr_2_super(linear_write_side_effect_in_pa
                                                                 (e))),
                                          (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
            (split(min_page, a′, bl′))
```

Using lemma every_conjunct_left,
Using lemma every_conjunct_left,
Using lemma every_conjunct_left,
Using lemma every_extend[[Address, list[Byte]], [Memory_Address_4G, list[Byte]]],
Using lemma every_conjunct_left,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 -4 -5 -6 -7 -8 -9 -12 -16),
Installing automatic rewrites from: (##! expr_2_super! expr_2_super_res!)
Using lemma every_implied,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 2),
Expanding the definition of linear_write_side_effect_in_page,
Repeatedly Skolemizing and flattening,
Case splitting on pm!1‘rw_addr(t!1‘1),
we get 2 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1:

{-1} pm′‘rw_addr(t′‘1)
{-2} cons?(t′‘2)
{-3} $a' \leq t'‘1$
{-4} $t'‘1 + \text{length}(t'‘2) \leq a' + \text{length}(\text{bl}')$
{-5} type_of(t′‘1) = type_of(a′)
{-6} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-7} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-8} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-9} transformer_invariant?(pm′‘states,
singleton(expr_2_super(λ (s:
Linear_memory[Physical_memory, pm_phy]):
(linear_resolve(t′‘1, Write) ##
(λ (pa: Memory_Address_4G)
(ns:
Linear_memory
[Physical_memory, pm_phy]):
memory_write_side_effect(pm_phy‘mem)
(pa, t′‘2, TRUE)(s)))
(s))))
{-10} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-11} Mem?(a′‘type_of)
{-12} 0 ≤ a′‘offset
{-13} a′‘offset < max_linear_offset
{-14} cons?[Byte](bl′)

{1} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super(λ (s:
Linear_memory
[Physical_memory, pm_phy]):
(linear_resolve(t′‘1, Write) ##
(λ
(pa: Memory_Address_4G)
(ns:
Linear_memory
[Physical_memory, pm_phy]):
memory_write_side_effect(pm_phy‘mem)
(pa, t′‘2, TRUE)(s)))
(s))),
(pm_phy‘ro_addr ∪ pm_phy‘rw_addr))

Using lemma pm_states,
Using lemma pm_plain_phy,
Expanding the definition of unchanged_memory_invariant?,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -13,

Hiding formulas: -13,

Using lemma pm_linear_resolve_write_ok,

Case splitting on subset?(address_block(data(linear_resolve(t!1‘1, Write)(s!1)), length(t!1‘2)), pm_phy‘rw_addr),

we get 3 subgoals:

```
split_linear_write_side_effects_unchanged_memory.1.1.1:

  {-1}  (address_block(data(linear_resolve(t'1, Write)(s')), length(t'2)) ⊆ pm_phy`rw_addr)
  {-2}  is_linear_plain_memory?(pm') ⊃ OK?(linear_resolve(t'1, Write)(s'))
  {-3}  plain_memory?(pm_phy)
  {-4}  pm'`states = pm_phy`states
  {-5}  pm'`rw_addr(t'1)
  {-6}  cons?(t'2)
  {-7}  a' ≤ t'1
  {-8}  t'1 + length(t'2) ≤ a' + length(bl')
  {-9}  type_of(t'1) = type_of(a')
  {-10} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'1
  {-11} t'1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
  {-12} rem(expt(2, min_page))(offset(t'1)) + length(t'2) ≤ expt(2, min_page)
  {-13} transformer_invariant?(pm'`states,
                               {y |
                                  y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                  CASES lin-
        ear_resolve(t'1, Write)(s) OF
                                                    OK(state, data):
                                                      mem-
        ory_write_side_effect(pm_phy`mem)
                                                            (data, t'2, TRUE)(s),
                                                    Exception(ex_type, state):
                                                      Exception(ex_type, state),
                                                    Fatal: Fatal,
                                                    Hang: Hang
                                                    ENDCASES)})
  {-14} pm'`states(s')
  {-15} union(pm_phy`ro_addr, pm_phy`rw_addr)(a'')
  {-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                           CASES linear_resolve(t'1, Write)(s) OF
                             OK(state, data):
                               memory_write_side_effect(pm_phy`mem)(data, t'2, TRUE)(s),
                             Exception(ex_type, state): Exception(ex_type, state),
                             Fatal: Fatal,
                             Hang: Hang
                             ENDCASES)
                         (s'))
  {-17} OK?(memory_read(pm_phy`mem)(a'')(s'))
  {-18} OK?(memory_read(pm_phy`mem)
                       (a'')
                       (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                             CASES linear_resolve(t'1, Write)(s) OF
                                               OK(state, data):
                                                 memory_write_side_effect(pm_phy`mem)
                                                       (data, t'2, TRUE)(s),
                                               Exception(ex_type, state): Excep-
        tion(ex_type, state),
                                               Fatal: Fatal,
                                               Hang: Hang
                                               ENDCASES)
                                           (s'))))
  {-19} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
  {-20} Mem?(a'`type_of)
  {-21} 0 ≤ a'`offset
  {-22} a'`offset < max_linear_offset                                              2299
  {-23} cons?[Byte](bl')
-------------------------------------------------------------------------------
   {1}  data(memory_read(pm_phy`mem)
                        (a'')
                        (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                              CASES linear_resolve(t'1, Write)(s) OF
                                                OK(state, data):
                                                  memory_write_side_effect(pm_phy`mem)
```

Using lemma linear_resolve_states,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`split_linear_write_side_effects_unchanged_memory.1.1.1.1:`

```
  {-1}  pm'`states(s')
  {-2}  OK?(linear_resolve(t'`1, Write)(s'))
  {-3}  pm_phy`states(state(linear_resolve(t'`1, Write)(s')))
  {-4}  (address_block(data(linear_resolve(t'`1, Write)(s')), length(t'`2)) ⊆ pm_phy`rw_addr)
  {-5}  plain_memory?(pm_phy)
  {-6}  pm'`states = pm_phy`states
  {-7}  pm'`rw_addr(t'`1)
  {-8}  cons?(t'`2)
  {-9}  a' ≤ t'`1
 {-10}  t'`1 + length(t'`2) ≤ a' + length(bl')
 {-11}  type_of(t'`1) = type_of(a')
 {-12}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
 {-13}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
 {-14}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
 {-15}  transformer_invariant?(pm'`states,
                               {y |
                                  y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                  CASES lin-
        ear_resolve(t'`1, Write)(s) OF
                                                    OK(state, data):
                                                      mem-
        ory_write_side_effect(pm_phy`mem)
                                                           (data, t'`2, TRUE)(s),
                                                    Exception(ex_type, state):
                                                      Exception(ex_type, state),
                                                    Fatal: Fatal,
                                                    Hang: Hang
                                                    ENDCASES)})
 {-16}  union(pm_phy`ro_addr, pm_phy`rw_addr)(a'')
 {-17}  OK?(memory_write_side_effect(pm_phy`mem)
                (data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)(s'))
 {-18}  OK?(OK(state(memory_write_side_effect(pm_phy`mem)
                           (data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)(s'))))
 {-19}  OK?(memory_read(pm_phy`mem)(a'')(s'))
 {-20}  OK?(memory_read(pm_phy`mem)
                (a'')
                (state(memory_write_side_effect(pm_phy`mem)
                           (data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)(s'))))
 {-21}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
 {-22}  Mem?(a'`type_of)
 {-23}  0 ≤ a'`offset
 {-24}  a'`offset < max_linear_offset
 {-25}  cons?[Byte](bl')
-------------------------------------------------------------------------------
  {1}   data(memory_read(pm_phy`mem)
                 (a'')
                 (state(memory_write_side_effect(pm_phy`mem)
                            (data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)
                            (s'))))
        = data(memory_read(pm_phy`mem)(a'')(s'))
```

Using lemma plain_memory_unchanged_invariant,

Using lemma unchanged_memory_invariant_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Case splitting on memory_write_side_effect_super_transformers(pm_phy‘mem, pm_phy‘rw_addr) (expr_2_super (memory_write_side_effect (pm_phy‘mem) (data (linear_resolve (t!1‘1, Write)(s!1)), t!1‘2, TRUE))),

we get 2 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1.1.1.1:

{-1} memory_write_side_effect_super_transformers(pm_phy`mem, pm_phy`rw_addr)
(expr_2_super(memory_write_side_effect(pm_phy`mem)
(data
(linear_resolve
(t'`1, Write)(s')),
t'`2,
TRUE)))
{-2} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-3} union(pm_phy`ro_addr, pm_phy`rw_addr)(a'')
{-4} OK?(memory_read(pm_phy`mem)(a'')(s'))
{-5} plain_memory?(pm_phy)
{-6} pm'`states(s')
{-7} OK?(linear_resolve(t'`1, Write)(s'))
{-8} pm_phy`states(state(linear_resolve(t'`1, Write)(s')))
{-9} (address_block(data(linear_resolve(t'`1, Write)(s')), length(t'`2)) ⊆ pm_phy`rw_addr)
{-10} pm'`states = pm_phy`states
{-11} pm'`rw_addr(t'`1)
{-12} cons?(t'`2)
{-13} a' ≤ t'`1
{-14} t'`1 + length(t'`2) ≤ a' + length(bl')
{-15} type_of(t'`1) = type_of(a')
{-16} reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-17} t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-18} rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-19} transformer_invariant?(pm'`states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES lin-
ear_resolve(t'`1, Write)(s) OF
OK(state, data):
mem-
ory_write_side_effect(pm_phy`mem)
(data, t'`2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-20} OK?(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)(s'))
{-21} OK?(OK(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)(s'))))
{-22} OK?(memory_read(pm_phy`mem)
(a'')
(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)(s'))))
{-23} (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-24} Mem?(a'`type_of)
{-25} 0 ≤ a'`offset
{-26} a'`offset < max_linear_offset
{-27} cons?[Byte](bl')

{1} union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_add
(memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)) ∪
(expr_2_super(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)))
{2} data(memory_read(pm_phy`mem)
(a'')
(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t'`1, Write)(s')), t'`2, TRUE)
(s'))))

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.1.1.1`.

split_linear_write_side_effects_unchanged_memory.1.1.1.1.2:

{-1} unchanged_memory_invariant?(pm_phy`mem, pm_phy`states,
((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_p
(pm_phy`ro_addr ∪ pm_phy`rw_addr))
{-2} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
{-3} OK?(memory_read(pm_phy`mem)(a″)(s′))
{-4} plain_memory?(pm_phy)
{-5} pm′`states(s′)
{-6} OK?(linear_resolve(t′`1, Write)(s′))
{-7} pm_phy`states(state(linear_resolve(t′`1, Write)(s′)))
{-8} (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
{-9} pm′`states = pm_phy`states
{-10} pm′`rw_addr(t′`1)
{-11} cons?(t′`2)
{-12} a′ ≤ t′`1
{-13} t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-14} type_of(t′`1) = type_of(a′)
{-15} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-16} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-17} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-18} transformer_invariant?(pm′`states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES lin-
ear_resolve(t′`1, Write)(s) OF
OK(state, data):
mem-
ory_write_side_effect(pm_phy`mem)
(data, t′`2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-19} OK?(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′))
{-20} OK?(OK(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′))))
{-21} OK?(memory_read(pm_phy`mem)
(a″)
(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)(s′))))
{-22} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-23} Mem?(a′`type_of)
{-24} 0 ≤ a′`offset
{-25} a′`offset < max_linear_offset
{-26} cons?[Byte](bl′)

{1} memory_write_side_effect_super_transformers(pm_phy`mem, pm_phy`rw_addr)
(expr_2_super(memory_write_side_effect(pm_phy`mem)
(data
(linear_resolve
(t′`1, Write)(s′)),
t′`2,
TRUE)))
{2} union((pm_phy`other_actions ∪ memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_add
(memory_read_side_effect_super_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)) ∪
(expr_2_super(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)))
{3} data(memory_read(pm_phy`mem)
(a″)
(state(memory_write_side_effect(pm_phy`mem)
(data(linear_resolve(t′`1, Write)(s′)), t′`2, TRUE)
(s′))))

Hiding formulas: (-1 -18 2 3),

Expanding the definition of memory_write_side_effect_super_transformers,

Instantiating quantified variables,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.1.1.2`.

split_linear_write_side_effects_unchanged_memory.1.1.1.2:

```
  {-1}  pm′‘states(s′)
  {-2}  OK?(linear_resolve(t′‘1, Write)(s′))
  {-3}  (address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)
  {-4}  plain_memory?(pm_phy)
  {-5}  pm′‘states = pm_phy‘states
  {-6}  pm′‘rw_addr(t′‘1)
  {-7}  cons?(t′‘2)
  {-8}  a′ ≤ t′‘1
  {-9}  t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
 {-10}  type_of(t′‘1) = type_of(a′)
 {-11}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
 {-12}  t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
 {-13}  rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
 {-14}  transformer_invariant?(pm′‘states,
                               {y |
                                  y =
                                   expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                  CASES lin-
        ear_resolve(t′‘1, Write)(s) OF
                                                    OK(state, data):
                                                      mem-
        ory_write_side_effect(pm_phy‘mem)
                                                           (data, t′‘2, TRUE)(s),
                                                    Exception(ex_type, state):
                                                      Exception(ex_type, state),
                                                    Fatal: Fatal,
                                                    Hang: Hang
                                                    ENDCASES)})
 {-15}  union(pm_phy‘ro_addr, pm_phy‘rw_addr)(a″)
 {-16}  OK?(memory_write_side_effect(pm_phy‘mem)
                (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))
 {-17}  OK?(OK(state(memory_write_side_effect(pm_phy‘mem)
                              (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))))
 {-18}  OK?(memory_read(pm_phy‘mem)(a″)(s′))
 {-19}  OK?(memory_read(pm_phy‘mem)
                (a″)
                (state(memory_write_side_effect(pm_phy‘mem)
                             (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)(s′))))
 {-20}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
 {-21}  Mem?(a′‘type_of)
 {-22}  0 ≤ a′‘offset
 {-23}  a′‘offset < max_linear_offset
 {-24}  cons?[Byte](bl′)
-------------------------------------------------------------------------------
  {1}   union(pm′‘ro_addr, pm′‘rw_addr)(t′‘1)
  {2}   data(memory_read(pm_phy‘mem)
                 (a″)
                 (state(memory_write_side_effect(pm_phy‘mem)
                              (data(linear_resolve(t′‘1, Write)(s′)), t′‘2, TRUE)
                              (s′))))
        = data(memory_read(pm_phy‘mem)(a″)(s′))
```

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.1.2`.

```
split_linear_write_side_effects_unchanged_memory.1.1.2:

  {-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
  {-2}  plain_memory?(pm_phy)
  {-3}  pm′`states = pm_phy`states
  {-4}  pm′`rw_addr(t′`1)
  {-5}  cons?(t′`2)
  {-6}  a′ ≤ t′`1
  {-7}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
  {-8}  type_of(t′`1) = type_of(a′)
  {-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
  {-10} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {-11} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
  {-12} transformer_invariant?(pm′`states,
                               {y |
                                 y =
                                  expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                 CASES lin-
        ear_resolve(t′`1, Write)(s) OF
                                                   OK(state, data):
                                                     mem-
        ory_write_side_effect(pm_phy`mem)
                                                           (data, t′`2, TRUE)(s),
                                                   Exception(ex_type, state):
                                                     Exception(ex_type, state),
                                                   Fatal: Fatal,
                                                   Hang: Hang
                                                   ENDCASES)})
  {-13} pm′`states(s′)
  {-14} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
  {-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                           CASES linear_resolve(t′`1, Write)(s) OF
                             OK(state, data):
                               memory_write_side_effect(pm_phy`mem)(data, t′`2, TRUE)(s),
                             Exception(ex_type, state): Exception(ex_type, state),
                             Fatal: Fatal,
                             Hang: Hang
                             ENDCASES)
                         (s′))
  {-16} OK?(memory_read(pm_phy`mem)(a″)(s′))
  {-17} OK?(memory_read(pm_phy`mem)
                       (a″)
                       (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                             CASES linear_resolve(t′`1, Write)(s) OF
                                               OK(state, data):
                                                 memory_write_side_effect(pm_phy`mem)
                                                       (data, t′`2, TRUE)(s),
                                               Exception(ex_type, state): Excep-
        tion(ex_type, state),
                                               Fatal: Fatal,
                                               Hang: Hang
                                               ENDCASES)
                                           (s′))))
  {-18} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
  {-19} Mem?(a′`type_of)
  {-20} 0 ≤ a′`offset
  {-21} a′`offset < max_linear_offset
  {-22} cons?[Byte](bl′)                                                    2309
-------------------------------------------------------------------------------
  {1}   (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
  {2}   data(memory_read(pm_phy`mem)
                        (a″)
                        (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                              CASES linear_resolve(t′`1, Write)(s) OF
                                                OK(state, data):
                                                  memory_write_side_effect(pm_phy`mem)
```

Hiding formulas: 2,

Using lemma same_page_address_block_write,

we get 3 subgoals:

split_linear_write_side_effects_unchanged_memory.1.1.2.1:

{-1} is_linear_plain_memory?(pm′) ∧
pm′‘states(s′) ∧
pm′‘rw_addr(t′‘1) ∧
a′ ≤ t′‘1 ∧
t′‘1 + length(t′‘2) ≤ a′ + length(bl′) ∧
rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
∧ (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
⊃
(address_block(data(linear_resolve(t′‘1, Write)(s′)), length(t′‘2)) ⊆ pm_phy‘rw_addr)

{-2} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))

{-3} plain_memory?(pm_phy)

{-4} pm′‘states = pm_phy‘states

{-5} pm′‘rw_addr(t′‘1)

{-6} cons?(t′‘2)

{-7} a′ ≤ t′‘1

{-8} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)

{-9} type_of(t′‘1) = type_of(a′)

{-10} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1

{-11} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))

{-12} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)

{-13} transformer_invariant?(pm′‘states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):
memory_write_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})

{-14} pm′‘states(s′)

{-15} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(a″)

{-16} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):
memory_write_side_effect(pm_phy‘mem)(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))

{-17} OK?(memory_read(pm_phy‘mem)(a″)(s′))

{-18} OK?(memory_read(pm_phy‘mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):
memory_write_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′)))))

{-19} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)

{-20} Mem?(a′‘type_of)

{-21} 0 ≤ a′‘offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.2.1`.

split_linear_write_side_effects_unchanged_memory.1.1.2.2:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  pm′`rw_addr(t′`1)
{-5}  cons?(t′`2)
{-6}  a′ ≤ t′`1
{-7}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-8}  type_of(t′`1) = type_of(a′)
{-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-10} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-11} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm′`states,
                             {y |
                                y =
                                 expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                CASES linear_resolve(t′`1, Write)(s) OF
                                                  OK(state, data):
                                                    memory_write_side_effect(pm_phy`mem)
                                                      (data, t′`2, TRUE)(s),
                                                  Exception(ex_type, state):
                                                    Exception(ex_type, state),
                                                  Fatal: Fatal,
                                                  Hang: Hang
                                                ENDCASES)})
{-13} pm′`states(s′)
{-14} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         CASES linear_resolve(t′`1, Write)(s) OF
                           OK(state, data):
                             memory_write_side_effect(pm_phy`mem)(data, t′`2, TRUE)(s),
                           Exception(ex_type, state): Exception(ex_type, state),
                           Fatal: Fatal,
                           Hang: Hang
                         ENDCASES)
                       (s′))
{-16} OK?(memory_read(pm_phy`mem)(a″)(s′))
{-17} OK?(memory_read(pm_phy`mem)
                     (a″)
                     (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                           CASES linear_resolve(t′`1, Write)(s) OF
                                             OK(state, data):
                                               memory_write_side_effect(pm_phy`mem)
                                                 (data, t′`2, TRUE)(s),
                                             Exception(ex_type, state): Exception(ex_type, state),
                                             Fatal: Fatal,
                                             Hang: Hang
                                           ENDCASES)
                                         (s′))))
{-18} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-19} Mem?(a′`type_of)
{-20} 0 ≤ a′`offset
{-21} a′`offset < max_linear_offset
{-22} cons?[Byte](bl′)                                                    2313
-------------------------------------------------------------------------------
 {1}  length[Byte](t′`2) > 0
 {2}  (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.2.2`.

split_linear_write_side_effects_unchanged_memory.1.1.2.3:

```
{-1}  is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′`1, Write)(s′))
{-2}  plain_memory?(pm_phy)
{-3}  pm′`states = pm_phy`states
{-4}  pm′`rw_addr(t′`1)
{-5}  cons?(t′`2)
{-6}  a′ ≤ t′`1
{-7}  t′`1 + length(t′`2) ≤ a′ + length(bl′)
{-8}  type_of(t′`1) = type_of(a′)
{-9}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′`1
{-10} t′`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-11} rem(expt(2, min_page))(offset(t′`1)) + length(t′`2) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm′`states,
                             {y |
                                y =
                                 expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                                CASES linear_resolve(t′`1, Write)(s) OF
                                                  OK(state, data):
                                                    memory_write_side_effect(pm_phy`mem)
                                                      (data, t′`2, TRUE)(s),
                                                  Exception(ex_type, state):
                                                    Exception(ex_type, state),
                                                  Fatal: Fatal,
                                                  Hang: Hang
                                                ENDCASES)})
{-13} pm′`states(s′)
{-14} union(pm_phy`ro_addr, pm_phy`rw_addr)(a″)
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                         CASES linear_resolve(t′`1, Write)(s) OF
                           OK(state, data):
                             memory_write_side_effect(pm_phy`mem)(data, t′`2, TRUE)(s),
                           Exception(ex_type, state): Exception(ex_type, state),
                           Fatal: Fatal,
                           Hang: Hang
                         ENDCASES)
                       (s′))
{-16} OK?(memory_read(pm_phy`mem)(a″)(s′))
{-17} OK?(memory_read(pm_phy`mem)
                     (a″)
                     (state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
                                           CASES linear_resolve(t′`1, Write)(s) OF
                                             OK(state, data):
                                               memory_write_side_effect(pm_phy`mem)
                                                 (data, t′`2, TRUE)(s),
                                             Exception(ex_type, state): Exception(ex_type, state),
                                             Fatal: Fatal,
                                             Hang: Hang
                                           ENDCASES)
                                         (s′))))
{-18} (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
{-19} Mem?(a′`type_of)
{-20} 0 ≤ a′`offset
{-21} a′`offset < max_linear_offset
{-22} cons?[Byte](bl′)                                                    2315
-----------------------------------------------------------------------------
 {1}  length[Byte](bl′) > 0
 {2}  (address_block(data(linear_resolve(t′`1, Write)(s′)), length(t′`2)) ⊆ pm_phy`rw_addr)
```

Expanding the definition of length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.2.3`.

split_linear_write_side_effects_unchanged_memory.1.1.3:

{-1} is_linear_plain_memory?(pm′) ⊃ OK?(linear_resolve(t′‘1, Write)(s′))
{-2} plain_memory?(pm_phy)
{-3} pm′‘states = pm_phy‘states
{-4} pm′‘rw_addr(t′‘1)
{-5} cons?(t′‘2)
{-6} a′ ≤ t′‘1
{-7} t′‘1 + length(t′‘2) ≤ a′ + length(bl′)
{-8} type_of(t′‘1) = type_of(a′)
{-9} reg_base((#type_of := Mem_, offset := 0#))(type_of(a′)) ≤ t′‘1
{-10} t′‘1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
{-11} rem(expt(2, min_page))(offset(t′‘1)) + length(t′‘2) ≤ expt(2, min_page)
{-12} transformer_invariant?(pm′‘states,
{y |
y =
expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):
memory_write_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state):
Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)})
{-13} pm′‘states(s′)
{-14} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(a″)
{-15} OK?(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):
memory_write_side_effect(pm_phy‘mem)(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))
{-16} OK?(memory_read(pm_phy‘mem)(a″)(s′))
{-17} OK?(memory_read(pm_phy‘mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):
memory_write_side_effect(pm_phy‘mem)
(data, t′‘2, TRUE)(s),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)
(s′))))
{-18} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-19} Mem?(a′‘type_of)
{-20} 0 ≤ a′‘offset
{-21} a′‘offset < max_linear_offset
{-22} cons?[Byte](bl′)

2317

{1} OK?[Linear_memory[Physical_memory, pm_phy], Address]
(linear_resolve[Physical_memory, pm_phy](t′‘1, Write)(s′))
{2} data(memory_read(pm_phy‘mem)
(a″)
(state(expr_2_super(λ (s: Linear_memory[Physical_memory, pm_phy]):
CASES linear_resolve(t′‘1, Write)(s) OF
OK(state, data):

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.1.3`.

`split_linear_write_side_effects_unchanged_memory.1.2`:

```
{-1}  cons?(t'`2)
{-2}  a' ≤ t'`1
{-3}  t'`1 + length(t'`2) ≤ a' + length(bl')
{-4}  type_of(t'`1) = type_of(a')
{-5}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ t'`1
{-6}  t'`1 < reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a'))
{-7}  rem(expt(2, min_page))(offset(t'`1)) + length(t'`2) ≤ expt(2, min_page)
{-8}  transformer_invariant?(pm'`states,
                             singleton(expr_2_super(λ (s:
                                                      Lin-
      ear_memory[Physical_memory, pm_phy]):
                                                        (linear_resolve(t'`1, Write) ##
                                                           (λ (pa: Mem-
      ory_Address_4G)
                                                              (ns:
                                                                 Linear_memory
                                                                 [Physical_memory, pm_
                                                               mem-
      ory_write_side_effect(pm_phy`mem)
                                                                    (pa, t'`2, TRUE)(s)))
                                                            (s))))
{-9}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-10} Mem?(a'`type_of)
{-11} 0 ≤ a'`offset
{-12} a'`offset < max_linear_offset
{-13} cons?[Byte](bl')
-------------------------------------------------------------------------------
 {1}  pm'`rw_addr(t'`1)
 {2}  unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                  singleton(expr_2_super(λ (s:
                                                            Linear_memory
                                                            [Physical_memory, pm_
                                                          (linear_resolve(t'`1, Write) #
                                                             (λ
                                                               (pa: Mem-
      ory_Address_4G)
                                                               (ns:
                                                                  Linear_memory
                                                                  [Physical_memory, pr
                                                                mem-
      ory_write_side_effect(pm_phy`mem)
                                                                  (pa, t'`2, TRUE)(s)))
                                                              (s))),
                                  (pm_phy`ro_addr ∪ pm_phy`rw_addr))
```

Hiding formulas: (-8 -7 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.1.2`.

`split_linear_write_side_effects_unchanged_memory.2:`

```
  {-1}  is_linear_plain_memory?(pm′)
  {-2}  (address_block(a′, length(bl′)) ⊆ pm′`rw_addr)
  {-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
                transformer_invariant?(pm′`states,
                                       singleton(expr_2_super(linear_write_side_effect_in_page(e)))))
              (split(min_page, a′, bl′))
  {-4}  every(λ (a_2: Address, bl2: list[Byte]):
                rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
              (split(min_page, a′, bl′))
  {-5}  every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a′, bl′))
  {-6}  every(λ (a_2: Address, bl2: list[Byte]):
                a′ ≤ a_2 ∧ a_2 + length(bl2) ≤ a′ + length(bl′))
              (split(min_page, a′, bl′))
  {-7}  Mem?(a′`type_of)
  {-8}  0 ≤ a′`offset
  {-9}  a′`offset < max_linear_offset
 {-10}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
              (bl′)
 {-11}  cons?[Byte](bl′)
--------------------------------------------------------------------------------
  {1}   a′ + length(bl′) ≤
         reg_size((#type_of := Mem_, offset := max_linear_offset#))(type_of(a′))
  {2}   every(λ (e: [Memory_Address_4G, list[Byte]]):
                unchanged_memory_invariant?(pm_phy`mem, pm′`states,
                                            singleton(expr_2_super(linear_write_side_effect_in_page
                                                                   (e))),
                                            (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
              (split(min_page, a′, bl′))
```

Hiding formulas: (-3 -4 -5 -6 -10 2),

Expanding the definition of length,

Using lemma pm_memory_addr,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Expanding the definition of subset?,

Installing automatic rewrites from: (Mem! max_linear! bus_width!)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_linear_write_side_effects_unchanged_memory.2`.

`split_linear_write_side_effects_unchanged_memory.3:`

```
{-1}  is_linear_plain_memory?(pm')
{-2}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-3}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              transformer_invariant?(pm'`states,
                                     singleton(expr_2_super(linear_write_side_effect_in_page(e))
            (split(min_page, a', bl'))
{-4}  every(λ (a_2: Address, bl2: list[Byte]):
              rem(expt(2, min_page))(offset(a_2)) + length(bl2) ≤ expt(2, min_page))
            (split(min_page, a', bl'))
{-5}  every(λ (a_2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-6}  every(λ (a_2: Address, bl2: list[Byte]):
              a' ≤ a_2 ∧ a_2 + length(bl2) ≤ a' + length(bl'))
            (split(min_page, a', bl'))
{-7}  Mem?(a'`type_of)
{-8}  0 ≤ a'`offset
{-9}  a'`offset < max_linear_offset
{-10} every(λ (x: number):
              number_field_pred(x) ∧
                real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl')
{-11} cons?[Byte](bl')
-------------------------------------------------------------------------------
 {1}  reg_base((#type_of := Mem_, offset := 0#))(type_of(a')) ≤ a'
 {2}  every(λ (e: [Memory_Address_4G, list[Byte]]):
              unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                          singleton(expr_2_super(linear_write_side_effect_in_pa
                                                                 (e))),
                                          (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
            (split(min_page, a', bl'))
```

Expanding the definition of reg_base,

Expanding the definition of <=,

which is trivially true.

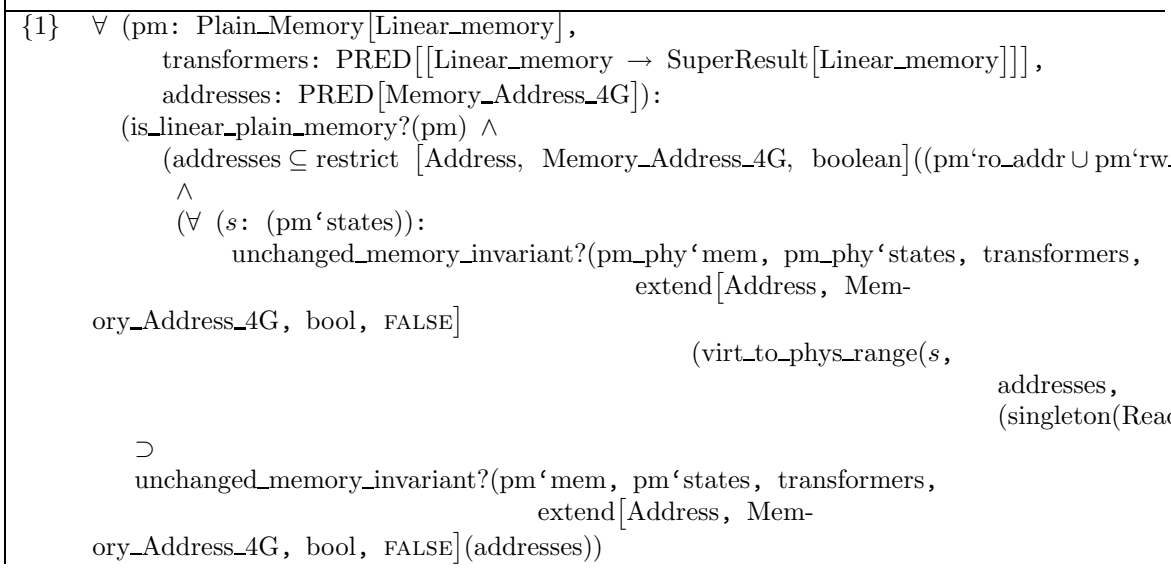
This completes the proof of `split_linear_write_side_effects_unchanged_memory.3`.

Q.E.D.

C.117.84 Linear_Memory_Properties.linear_unchanged_invariant

Terse proof for `linear_unchanged_invariant`.

`linear_unchanged_invariant:`



Installing automatic rewrites from: has_next_state fatal_result

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Using lemma pm_states,

Case splitting on transformer_invariant?(pm!1‘states, transformers!1),

we get 2 subgoals:

`linear_unchanged_invariant.1:`

{-1} transformer_invariant?(pm′‘states, transformers′)
{-2} is_linear_plain_memory?(pm′) ⊃ pm′‘states = pm_phy‘states
{-3} is_linear_plain_memory?(pm′)
{-4} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-5} ∀ (s: (pm′‘states)):
transformer_invariant?(pm_phy‘states, transformers′) ∧
(∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
a: Address):
pm_phy‘states(s_1) ∧
transformers′(q) ∧
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s, addresses′,
(singleton(Read) ∪ singleton(Execute))))
(a)
∧
OK?(q(s_1)) ∧
OK?(memory_read(pm_phy‘mem)(a)(s_1)) ∧
OK?(memory_read(pm_phy‘mem)(a)(state(q(s_1))))
⊃
data(memory_read(pm_phy‘mem)(a)(state(q(s_1)))) =
data(memory_read(pm_phy‘mem)(a)(s_1)))

{1} transformer_invariant?(pm′‘states, transformers′) ∧
(∀ (s: Linear_memory[Physical_memory, pm_phy],
q: [Linear_memory[Physical_memory, pm_phy] → SuperResult[Physical_memory]],
a: Address):
pm′‘states(s) ∧
transformers′(q) ∧
extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′‘mem)(a)(s)) ∧
OK?(memory_read(pm′‘mem)(a)(state(q(s))))
⊃
data(memory_read(pm′‘mem)(a)(state(q(s)))) =
data(memory_read(pm′‘mem)(a)(s)))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Rewriting using pm_read_linear, matching in *,

Expanding the definition of linear_read,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##! singleton!)

Case splitting on union(pm!1‘ro_addr, pm!1‘rw_addr)(a!1),

we get 2 subgoals:

linear_unchanged_invariant.1.1:

```
{-1}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-2}  transformer_invariant?(pm'`states, transformers')
{-3}  is_linear_plain_memory?(pm')
{-4}  pm'`states = pm_phy`states
{-5}  (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
{-6}  ∀ (s: (pm'`states)):
        ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
             a: Address):
          pm_phy`states(s_1) ∧
           transformers'(q) ∧
            extend[Address, Memory_Address_4G, bool, FALSE]
                (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
                (a)
              ∧
              OK?(q(s_1)) ∧
               OK?(memory_read(pm_phy`mem)(a)(s_1)) ∧
                OK?(memory_read(pm_phy`mem)(a)(state(q(s_1))))
             ⊃
             data(memory_read(pm_phy`mem)(a)(state(q(s_1)))) =
              data(memory_read(pm_phy`mem)(a)(s_1))
{-7}  pm'`states(s')
{-8}  transformers'(q')
{-9}  extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-10} OK?(q'(s'))
{-11} OK?(IF in_memory(min_linear, max_linear)(a')
            THEN IF Mem?(type_of(a'))
                    THEN (linear_resolve(a', Read) ##
                            (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                             (s')
                    ELSE memory_read(pm_phy`mem)(a')(s')
                    ENDIF
            ELSE fatal_result(s')
            ENDIF)
{-12} OK?(IF in_memory(min_linear, max_linear)(a')
            THEN IF Mem?(type_of(a'))
                    THEN (linear_resolve(a', Read) ##
                            (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                             (state(q'(s')))
                    ELSE memory_read(pm_phy`mem)(a')(state(q'(s')))
                    ENDIF
            ELSE fatal_result(state(q'(s')))
            ENDIF)
-------------------------------------------------------------------------------
{1}   IF in_memory(min_linear, max_linear)(a')
        THEN IF Mem?(type_of(a'))
                THEN data((linear_resolve(a', Read) ##
                            (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                             (state(q'(s'))))
                ELSE data(memory_read(pm_phy`mem)(a')(state(q'(s'))))
                ENDIF
      ELSE data(fatal_result(state(q'(s'))))
      ENDIF
       =
       IF in_memory(min_linear, max_linear)(a')
         THEN IF Mem?(type_of(a'))
                 THEN data((linear_resolve(a', Read) ##
                             (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                              (s'))
                 ELSE data(memory_read(pm_phy`mem)(a')(s'))
                 ENDIF
       ELSE data(fatal_result(s'))
       ENDIF
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`linear_unchanged_invariant.1.1.1:`

```
{-1}  union(pm'‘ro_addr, pm'‘rw_addr)(a')
{-2}  transformer_invariant?(pm'‘states, transformers')
{-3}  is_linear_plain_memory?(pm')
{-4}  pm'‘states = pm_phy‘states
{-5}  (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'‘ro_addr ∪ pm'‘rw_addr)))
{-6}  ∀ (s: (pm'‘states)):
        ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
           a: Address):
          pm_phy‘states(s_1) ∧
           transformers'(q) ∧
            extend[Address, Memory_Address_4G, bool, FALSE]
                 (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
                 (a)
              ∧
              OK?(q(s_1)) ∧
               OK?(memory_read(pm_phy‘mem)(a)(s_1)) ∧
                OK?(memory_read(pm_phy‘mem)(a)(state(q(s_1))))
            ⊃
            data(memory_read(pm_phy‘mem)(a)(state(q(s_1)))) =
             data(memory_read(pm_phy‘mem)(a)(s_1))
{-7}  pm'‘states(s')
{-8}  transformers'(q')
{-9}  extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-10} OK?(q'(s'))
{-11} in_memory(min_linear, max_linear)(a')
{-12} Mem?(type_of(a'))
{-13} OK?(linear_resolve(a', Read)(s'))
{-14} OK?(memory_read(pm_phy‘mem)
                (data(linear_resolve(a', Read)(s')))
                (state(linear_resolve(a', Read)(s'))))
{-15} OK?(linear_resolve(a', Read)(state(q'(s'))))
{-16} OK?(memory_read(pm_phy‘mem)
                (data(linear_resolve(a', Read)(state(q'(s')))))
                (state(linear_resolve(a', Read)(state(q'(s'))))))
-------------------------------------------------------------------------------
 {1}  data(memory_read(pm_phy‘mem)
                 (data(linear_resolve(a', Read)(state(q'(s')))))
                 (state(linear_resolve(a', Read)(state(q'(s'))))))
       =
      data(memory_read(pm_phy‘mem)
                  (data(linear_resolve(a', Read)(s')))
                  (state(linear_resolve(a', Read)(s'))))
```

Using lemma super_transformer_invariant_next_ok,
Using lemma linear_resolve_same_result,
Using lemma linear_resolve_unchanged_pm_phy,
Using lemma unchanged_memory_invariant_unchanged,
Using lemma unchanged_memory_invariant_unchanged,
Case splitting on union(pm_phy‘ro_addr, pm_phy‘rw_addr) (data(linear_resolve(a!1, Read)(s!1))),
we get 2 subgoals:

linear_unchanged_invariant.1.1.1.1:

{-1} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))

{-2} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(state(q′(s′))) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(state(q′(s′)))) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′)))))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))

{-3} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(s′) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(s′)) ∧
OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′)) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
= data(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))

{-4} union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)
⊃
unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm′‘states,

Case splitting on OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a!1, Read)(s!1)))(s!1)), OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a!1, Read)(s!1)))(state(q!1(s!1)))),

we get 3 subgoals:

linear_unchanged_invariant.1.1.1.1.1:

{-1} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-2} OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-3} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(state(q′(s′))) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(state(q′(s′)))) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-5} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(s′) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(s′)) ∧
OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′)) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
= data(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-6} union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

linear_unchanged_invariant.1.1.1.1.1.1:

{-1} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-2} OK?(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-3} union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-4} unchanged_memory_invariant?(pm_phy`mem, pm′`states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ extend [Address, Memory_Addres
{-5} pm′`states(state(q′(s′)))
{-6} data(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(state(q′(s′))))))
=
data(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-7} pm′`states(s′)
{-8} data(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
= data(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-9} union(pm′`ro_addr, pm′`rw_addr)(a′)
{-10} is_linear_plain_memory?(pm′)
{-11} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-12} OK?(linear_resolve(a′, Read)(s′))
{-13} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-14} transformer_invariant?(pm′`states, transformers′)
{-15} transformers′(q′)
{-16} pm′`states = pm_phy`states
{-17} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)))
{-18} ∀ (s: (pm′`states)):
∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
a: Address):
pm_phy`states(s_1) ∧
transformers′(q) ∧
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s, addresses′, (singleton(Read) ∪ singleton(Execute))))
(a)
∧
OK?(q(s_1)) ∧
OK?(memory_read(pm_phy`mem)(a)(s_1)) ∧
OK?(memory_read(pm_phy`mem)(a)(state(q(s_1))))
⊃
data(memory_read(pm_phy`mem)(a)(state(q(s_1)))) =
data(memory_read(pm_phy`mem)(a)(s_1))
{-19} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a′)
{-20} OK?(q′(s′))
{-21} in_memory(min_linear, max_linear)(a′)
{-22} Mem?(type_of(a′))
{-23} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-24} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} data(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))
=
data(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))

Replacing using formula -8,

Replacing using formula -13,

Instantiating the top quantifier in -18 with the terms: (s!1),

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-4 2),

Expanding the definition of extend,

Expanding the definition of virt_to_phys_range,

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.1`.

linear_unchanged_invariant.1.1.1.1.1.2:

{-1} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-2} OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-3} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
{-5} pm′‘states(state(q′(s′)))
{-6} pm′‘states(s′)
{-7} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-8} is_linear_plain_memory?(pm′)
{-9} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-10} OK?(linear_resolve(a′, Read)(s′))
{-11} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-12} transformer_invariant?(pm′‘states, transformers′)
{-13} transformers′(q′)
{-14} pm′‘states = pm_phy‘states
{-15} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-16} ∀ (s: (pm′‘states)):
∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
a: Address):
pm_phy‘states(s_1) ∧
transformers′(q) ∧
extend[Address, Memory_Address_4G, bool, FALSE]
(virt_to_phys_range(s, addresses′, (singleton(Read) ∪ singleton(Execute))))
(a)
∧
OK?(q(s_1)) ∧
OK?(memory_read(pm_phy‘mem)(a)(s_1)) ∧
OK?(memory_read(pm_phy‘mem)(a)(state(q(s_1))))
⊃
data(memory_read(pm_phy‘mem)(a)(state(q(s_1)))) =
data(memory_read(pm_phy‘mem)(a)(s_1))
{-17} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a′)
{-18} OK?(q′(s′))
{-19} in_memory(min_linear, max_linear)(a′)
{-20} Mem?(type_of(a′))
{-21} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-22} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
{2} data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))

Hiding formulas: (-4 -16 2),

Expanding the definition of difference,

Expanding the definition of member,

Expanding the definition of extend,

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -3 -4 -5 -7),

Expanding the definition of disjoint?,

Expanding the definition of empty?,

Expanding the definition of intersection,

Expanding the definition of member,

Instantiating the top quantifier in -1 with the terms: (data(linear_resolve(a!1, Read)(s!1))),

Case splitting on restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)) = union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)),

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.1.1.2.1:`

{-1} restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)) =
(restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Address, Memory_Address_4G,
{-2} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-3} OK?(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-4} union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-5} pm′`states(state(q′(s′)))
{-6} Mem?(data(linear_resolve(a′, Read)(s′))`type_of)
{-7} 0 ≤ data(linear_resolve(a′, Read)(s′))`offset
{-8} data(linear_resolve(a′, Read)(s′))`offset < max_linear_offset
{-9} address_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
(data(linear_resolve(a′, Read)(s′)))
{-10} pm′`states(s′)
{-11} union(pm′`ro_addr, pm′`rw_addr)(a′)
{-12} is_linear_plain_memory?(pm′)
{-13} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-14} OK?(linear_resolve(a′, Read)(s′))
{-15} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-16} transformer_invariant?(pm′`states, transformers′)
{-17} transformers′(q′)
{-18} pm′`states = pm_phy`states
{-19} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)))
{-20} 0 ≤ a′`offset
{-21} a′`offset < max_linear_offset
{-22} addresses′(a′)
{-23} OK?(q′(s′))
{-24} in_memory(min_linear, max_linear)(a′)
{-25} Mem?(type_of(a′))
{-26} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-27} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} virt_to_phys_range(s′,
(restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Address,
(data(linear_resolve(a′, Read)(s′)))
∧
address_in_pt_range?(s′,
(restrict [Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Add
(data(linear_resolve(a′, Read)(s′)))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of virt_to_phys_range,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.1.1.2.1.1:`

{-1}	restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)) = (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_A
{-2}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-3}	OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-4}	union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-5}	pm′‘states(state(q′(s′)))
{-6}	Mem?(data(linear_resolve(a′, Read)(s′))‘type_of)
{-7}	0 ≤ data(linear_resolve(a′, Read)(s′))‘offset
{-8}	data(linear_resolve(a′, Read)(s′))‘offset < max_linear_offset
{-9}	address_in_pt_range?(s′, restrict[Address, Memory_Address_4G, boolean] ((pm′‘ro_addr ∪ pm′‘rw_addr))) (data(linear_resolve(a′, Read)(s′)))
{-10}	pm′‘states(s′)
{-11}	union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-12}	is_linear_plain_memory?(pm′)
{-13}	OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-14}	OK?(linear_resolve(a′, Read)(s′))
{-15}	data(linear_resolve(a′, Read)(state(q′(s′)))) = data(linear_resolve(a′, Read)(s′))
{-16}	transformer_invariant?(pm′‘states, transformers′)
{-17}	transformers′(q′)
{-18}	pm′‘states = pm_phy‘states
{-19}	(addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-20}	0 ≤ a′‘offset
{-21}	a′‘offset < max_linear_offset
{-22}	addresses′(a′)
{-23}	OK?(q′(s′))
{-24}	in_memory(min_linear, max_linear)(a′)
{-25}	Mem?(type_of(a′))
{-26}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a′, Read)(s′))) (state(linear_resolve(a′, Read)(s′))))
{-27}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a′, Read)(state(q′(s′))))) (state(linear_resolve(a′, Read)(state(q′(s′))))))
{1}	fullset[Memory_access](Read)

Expanding the definition of fullset,

which is trivially true.

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.2.1.1`.

`linear_unchanged_invariant.1.1.1.1.1.2.1.2:`

{-1} restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)) =
(restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr) ∪ restrict [Address, Memory_Address_4G,
{-2} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-3} OK?(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-4} union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-5} pm′`states(state(q′(s′)))
{-6} Mem?(data(linear_resolve(a′, Read)(s′))`type_of)
{-7} 0 ≤ data(linear_resolve(a′, Read)(s′))`offset
{-8} data(linear_resolve(a′, Read)(s′))`offset < max_linear_offset
{-9} address_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′`ro_addr ∪ pm′`rw_addr)))
(data(linear_resolve(a′, Read)(s′)))
{-10} pm′`states(s′)
{-11} union(pm′`ro_addr, pm′`rw_addr)(a′)
{-12} is_linear_plain_memory?(pm′)
{-13} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-14} OK?(linear_resolve(a′, Read)(s′))
{-15} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-16} transformer_invariant?(pm′`states, transformers′)
{-17} transformers′(q′)
{-18} pm′`states = pm_phy`states
{-19} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)))
{-20} 0 ≤ a′`offset
{-21} a′`offset < max_linear_offset
{-22} addresses′(a′)
{-23} OK?(q′(s′))
{-24} in_memory(min_linear, max_linear)(a′)
{-25} Mem?(type_of(a′))
{-26} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-27} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} union[Memory_Address_4G]
(restrict[Address, Memory_Address_4G, boolean](pm′`ro_addr),
restrict[Address, Memory_Address_4G, boolean](pm′`rw_addr))
(a′)

Keeping (-11 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.2.1.2`.

`linear_unchanged_invariant.1.1.1.1.1.2.2:`

{-1} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-2} OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-3} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-4} pm′‘states(state(q′(s′)))
{-5} Mem?(data(linear_resolve(a′, Read)(s′))‘type_of)
{-6} 0 ≤ data(linear_resolve(a′, Read)(s′))‘offset
{-7} data(linear_resolve(a′, Read)(s′))‘offset < max_linear_offset
{-8} address_in_pt_range?(s′,
restrict[Address, Memory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))
(data(linear_resolve(a′, Read)(s′)))
{-9} pm′‘states(s′)
{-10} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-11} is_linear_plain_memory?(pm′)
{-12} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-13} OK?(linear_resolve(a′, Read)(s′))
{-14} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-15} transformer_invariant?(pm′‘states, transformers′)
{-16} transformers′(q′)
{-17} pm′‘states = pm_phy‘states
{-18} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-19} 0 ≤ a′‘offset
{-20} a′‘offset < max_linear_offset
{-21} addresses′(a′)
{-22} OK?(q′(s′))
{-23} in_memory(min_linear, max_linear)(a′)
{-24} Mem?(type_of(a′))
{-25} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-26} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)) =
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_A
{2} virt_to_phys_range(s′,
(restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict
(data(linear_resolve(a′, Read)(s′)))
∧
address_in_pt_range?(s′,
(restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ re
(data(linear_resolve(a′, Read)(s′)))

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.2.2`.

`linear_unchanged_invariant.1.1.1.1.1.3:`

```
{-1}  OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{-2}  OK?(memory_read(pm_phy`mem)(data(linear_resolve(a', Read)(s')))(s'))
{-3}  union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a', Read)(s')))
{-4}  unchanged_memory_invariant?(pm_phy`mem, pm'`states,
                                  singleton(expr_2_super[Physical_memory, Address]
                                                        (linear_resolve(a', Read))),
                                  ((pm_phy`ro_addr ∪ pm_phy`rw_addr) \ extend [Address, Memory_Addres
{-5}  pm'`states(state(q'(s')))
{-6}  pm'`states(s')
{-7}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-8}  is_linear_plain_memory?(pm')
{-9}  OK?(linear_resolve(a', Read)(state(q'(s'))))
{-10} OK?(linear_resolve(a', Read)(s'))
{-11} data(linear_resolve(a', Read)(state(q'(s')))) =
       data(linear_resolve(a', Read)(s'))
{-12} transformer_invariant?(pm'`states, transformers')
{-13} transformers'(q')
{-14} pm'`states = pm_phy`states
{-15} (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
{-16} ∀ (s: (pm'`states)):
        ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
           a: Address):
          pm_phy`states(s_1) ∧
           transformers'(q) ∧
            extend[Address, Memory_Address_4G, bool, FALSE]
                  (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
                  (a)
             ∧
             OK?(q(s_1)) ∧
              OK?(memory_read(pm_phy`mem)(a)(s_1)) ∧
               OK?(memory_read(pm_phy`mem)(a)(state(q(s_1))))
          ⊃
          data(memory_read(pm_phy`mem)(a)(state(q(s_1)))) =
           data(memory_read(pm_phy`mem)(a)(s_1))
{-17} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-18} OK?(q'(s'))
{-19} in_memory(min_linear, max_linear)(a')
{-20} Mem?(type_of(a'))
{-21} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(s')))
                (state(linear_resolve(a', Read)(s'))))
{-22} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(state(q'(s')))))
                (state(linear_resolve(a', Read)(state(q'(s'))))))
-------------------------------------------------------------------------
 {1}  expr_2_super(linear_resolve(a', Read)) =
       expr_2_super[Physical_memory, Address](linear_resolve(a', Read))
 {2}  data(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(state(q'(s')))))
                (state(linear_resolve(a', Read)(state(q'(s'))))))
      =
      data(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(s')))
                (state(linear_resolve(a', Read)(s'))))
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.1.3`.

linear_unchanged_invariant.1.1.1.1.2:

{-1} OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-3} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(state(q′(s′))) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(state(q′(s′)))) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
{-4} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(s′) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(s′)) ∧
OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′)) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
= data(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))
{-5} union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)
⊃
unchanged_memory_invariant?[Physical_memory]

Hiding formulas: (-3 -4 -5 -13 2),

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.1.2.1:`

{-1}	transformers_ok?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_a
{-2}	OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a', Read)(s')))(s'))
{-3}	union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a', Read)(s')))
{-4}	union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-5}	is_linear_plain_memory?(pm′)
{-6}	pm′‘states(state(q'(s')))
{-7}	pm′‘states(s')
{-8}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-9}	OK?(linear_resolve(a', Read)(s'))
{-10}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-11}	transformer_invariant?(pm′‘states, transformers′)
{-12}	transformers′(q')
{-13}	pm′‘states = pm_phy‘states
{-14}	(addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-15}	extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a')
{-16}	OK?(q'(s'))
{-17}	in_memory(min_linear, max_linear)(a')
{-18}	Mem?(type_of(a'))
{-19}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a', Read)(s'))) (state(linear_resolve(a', Read)(s'))))
{-20}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a', Read)(state(q'(s'))))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)) (expr_2_super(memory_read(pm_phy‘mem) (data(linear_resolve(a', Read)(s')))))
{2}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a', Read)(s')))(state(q'(s'))))

Using lemma memory_read_transformers_memory_read,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.2.1`.

`linear_unchanged_invariant.1.1.1.1.2.2:`

{-1} OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a', Read)(s')))(s'))
{-2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a', Read)(s')))
{-3} union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-4} is_linear_plain_memory?(pm′)
{-5} pm′‘states(state(q'(s')))
{-6} pm′‘states(s')
{-7} OK?(linear_resolve(a', Read)(state(q'(s'))))
{-8} OK?(linear_resolve(a', Read)(s'))
{-9} data(linear_resolve(a', Read)(state(q'(s')))) =
data(linear_resolve(a', Read)(s'))
{-10} transformer_invariant?(pm′‘states, transformers′)
{-11} transformers′(q')
{-12} pm′‘states = pm_phy‘states
{-13} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-14} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a')
{-15} OK?(q'(s'))
{-16} in_memory(min_linear, max_linear)(a')
{-17} Mem?(type_of(a'))
{-18} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)(s')))
(state(linear_resolve(a', Read)(s'))))
{-19} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)(state(q'(s')))))
(state(linear_resolve(a', Read)(state(q'(s'))))))

{1} transformers_ok?(pm_phy‘states,
memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_addr)))
{2} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)(s')))(state(q'(s'))))
{3} plain_memory?(pm_phy)

Using lemma pm_plain_phy,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.2.2`.

`linear_unchanged_invariant.1.1.1.1.3:`

{-1} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))

{-2} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
∧
pm′‘states(state(q′(s′))) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(state(q′(s′)))) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))

{-3} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Mem
∧
pm′‘states(s′) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(s′)) ∧
OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′)) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
= data(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))

{-4} union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)
⊃
unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm′‘states,

Hiding formulas: (-2 -3 -4 -12 2),

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.1.3.1:`

{-1} transformers_ok?(pm_phy`states, memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr)))
{-2} union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a′, Read)(s′)))
{-3} union(pm′`ro_addr, pm′`rw_addr)(a′)
{-4} is_linear_plain_memory?(pm′)
{-5} pm′`states(state(q′(s′)))
{-6} pm′`states(s′)
{-7} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-8} OK?(linear_resolve(a′, Read)(s′))
{-9} data(linear_resolve(a′, Read)(state(q′(s′)))) = data(linear_resolve(a′, Read)(s′))
{-10} transformer_invariant?(pm′`states, transformers′)
{-11} transformers′(q′)
{-12} pm′`states = pm_phy`states
{-13} (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)))
{-14} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a′)
{-15} OK?(q′(s′))
{-16} in_memory(min_linear, max_linear)(a′)
{-17} Mem?(type_of(a′))
{-18} OK?(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))(state(linear_resolve(a′, Read)(s′))))
{-19} OK?(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(state(q′(s′)))))(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} memory_read_transformers(pm_phy`mem, (pm_phy`ro_addr ∪ pm_phy`rw_addr))(expr_2_super(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))))
{2} OK?(memory_read(pm_phy`mem)(data(linear_resolve(a′, Read)(s′)))(s′))

Using lemma memory_read_transformers_memory_read,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.3.1`.

`linear_unchanged_invariant.1.1.1.1.3.2:`

{-1}	union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a', Read)(s')))
{-2}	union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-3}	is_linear_plain_memory?(pm′)
{-4}	pm′‘states(state($q'(s')$))
{-5}	pm′‘states(s')
{-6}	OK?(linear_resolve(a', Read)(state($q'(s')$)))
{-7}	OK?(linear_resolve(a', Read)(s'))
{-8}	data(linear_resolve(a', Read)(state($q'(s')$))) = data(linear_resolve(a', Read)(s'))
{-9}	transformer_invariant?(pm′‘states, transformers′)
{-10}	transformers′(q')
{-11}	pm′‘states = pm_phy‘states
{-12}	(addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-13}	extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a')
{-14}	OK?($q'(s')$)
{-15}	in_memory(min_linear, max_linear)(a')
{-16}	Mem?(type_of(a'))
{-17}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a', Read)(s'))) (state(linear_resolve(a', Read)(s'))))
{-18}	OK?(memory_read(pm_phy‘mem) (data(linear_resolve(a', Read)(state($q'(s')$)))) (state(linear_resolve(a', Read)(state($q'(s')$)))))
{1}	transformers_ok?(pm_phy‘states, memory_read_transformers(pm_phy‘mem, (pm_phy‘ro_addr ∪ pm_phy‘rw_a
{2}	OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a', Read)(s')))(s'))
{3}	plain_memory?(pm_phy)

Using lemma pm_plain_phy,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.1.3.2`.

linear_unchanged_invariant.1.1.1.2:

{-1} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(state(q′(s′))) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(state(q′(s′)))) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))
∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(state(q′(s′))))))
=
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))(state(q′(s′))))

{-2} unchanged_memory_invariant?(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address]
(linear_resolve(a′, Read))),
((pm_phy‘ro_addr ∪ pm_phy‘rw_addr) \ extend [Address, Memory_Addres
∧
pm′‘states(s′) ∧
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read)))
(expr_2_super(linear_resolve(a′, Read)))
∧
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s′,
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(data(linear_resolve(a′, Read)(s′)))
∧
OK?(expr_2_super(linear_resolve(a′, Read))(s′)) ∧
OK?(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′)) ∧
OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
⊃
data(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(expr_2_super(linear_resolve(a′, Read))(s′))))
= data(memory_read(pm_phy‘mem)(data(linear_resolve(a′, Read)(s′)))(s′))

{-3} union(pm′‘ro_addr, pm′‘rw_addr)(a′) ∧ is_linear_plain_memory?(pm′) ∧ pm′‘states(s′)
⊃
unchanged_memory_invariant?[Physical_memory]
(pm_phy‘mem, pm′‘states,
singleton(expr_2_super[Physical_memory, Address](linear_resolve(a′, Read))),

Hiding formulas: (-1 -2 -3 -11 2),

Using lemma pm_linear_blessed,

Expanding the definition of linear_blessed?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: (-1 -2 -4 -5 -6 -7),

Expanding the definition of subset?,

Instantiating the top quantifier in -1 with the terms: (data(linear_resolve(a!1, Read)(s!1))),

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.2.1:`

{-1}	(data(linear_resolve(a', Read)(s')) ∈ virt_to_phys_range(s', (restrict [Address, Memory_Address ⇒ (data(linear_resolve(a', Read)(s')) ∈ restrict [Address, Memory_Address_4G, boolean]((pm_ph
{-2}	union(pm′`ro_addr, pm′`rw_addr)(a')
{-3}	is_linear_plain_memory?(pm′)
{-4}	pm′`states(state(q'(s')))
{-5}	pm′`states(s')
{-6}	OK?(linear_resolve(a', Read)(state(q'(s'))))
{-7}	OK?(linear_resolve(a', Read)(s'))
{-8}	data(linear_resolve(a', Read)(state(q'(s')))) = data(linear_resolve(a', Read)(s'))
{-9}	transformer_invariant?(pm′`states, transformers′)
{-10}	transformers′(q')
{-11}	pm′`states = pm_phy`states
{-12}	∀ (x: Memory_Address_4G): (x ∈ addresses′) ⇒ (x ∈ restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr)))
{-13}	extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a')
{-14}	OK?(q'(s'))
{-15}	in_memory(min_linear, max_linear)(a')
{-16}	Mem?(type_of(a'))
{-17}	OK?(memory_read(pm_phy`mem) (data(linear_resolve(a', Read)(s'))) (state(linear_resolve(a', Read)(s'))))
{-18}	OK?(memory_read(pm_phy`mem) (data(linear_resolve(a', Read)(state(q'(s'))))) (state(linear_resolve(a', Read)(state(q'(s'))))))
{1}	union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a', Read)(s')))

Expanding the definition of member,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.2.1.1:`

{-1} restrict[Address, Memory_Address_4G, boolean]
((pm_phy`ro_addr ∪ pm_phy`rw_addr))(data(linear_resolve(a′, Read)(s′)))
{-2} union(pm′`ro_addr, pm′`rw_addr)(a′)
{-3} is_linear_plain_memory?(pm′)
{-4} pm′`states(state(q′(s′)))
{-5} pm′`states(s′)
{-6} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-7} OK?(linear_resolve(a′, Read)(s′))
{-8} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-9} transformer_invariant?(pm′`states, transformers′)
{-10} transformers′(q′)
{-11} pm′`states = pm_phy`states
{-12} ∀ (x: Memory_Address_4G):
addresses′(x) ⇒
restrict[Address, Memory_Address_4G, boolean]((pm′`ro_addr ∪ pm′`rw_addr))(x)
{-13} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a′)
{-14} OK?(q′(s′))
{-15} in_memory(min_linear, max_linear)(a′)
{-16} Mem?(type_of(a′))
{-17} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-18} OK?(memory_read(pm_phy`mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a′, Read)(s′)))

Expanding the definition of restrict,

which is trivially true.

This completes the proof of `linear_unchanged_invariant.1.1.1.2.1.1`.

`linear_unchanged_invariant.1.1.1.2.1.2:`

```
{-1}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states(state(q'(s')))
{-4}  pm'`states(s')
{-5}  OK?(linear_resolve(a', Read)(state(q'(s'))))
{-6}  OK?(linear_resolve(a', Read)(s'))
{-7}  data(linear_resolve(a', Read)(state(q'(s')))) =
        data(linear_resolve(a', Read)(s'))
{-8}  transformer_invariant?(pm'`states, transformers')
{-9}  transformers'(q')
{-10} pm'`states = pm_phy`states
{-11} ∀ (x: Memory_Address_4G):
        addresses'(x) ⇒
          restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr))(x)
{-12} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-13} OK?(q'(s'))
{-14} in_memory(min_linear, max_linear)(a')
{-15} Mem?(type_of(a'))
{-16} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(s')))
                (state(linear_resolve(a', Read)(s'))))
{-17} OK?(memory_read(pm_phy`mem)
                (data(linear_resolve(a', Read)(state(q'(s')))))
                (state(linear_resolve(a', Read)(state(q'(s'))))))
-------------------------------------------------------------------------------
 {1}  virt_to_phys_range(s',
                                (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr) ∪ restrict
                                (singleton(Read) ∪ singleton(Execute)))
                              (data(linear_resolve(a', Read)(s')))
 {2}  union(pm_phy`ro_addr, pm_phy`rw_addr)(data(linear_resolve(a', Read)(s')))
```

Hiding formulas: 2,

Expanding the definition of virt_to_phys_range,

Instantiating quantified variables,

we get 2 subgoals:

`linear_unchanged_invariant.1.1.1.2.1.2.1:`

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(a')
{-2} is_linear_plain_memory?(pm′)
{-3} pm′‘states(state($q'(s')$))
{-4} pm′‘states(s')
{-5} OK?(linear_resolve(a', Read)(state($q'(s')$)))
{-6} OK?(linear_resolve(a', Read)(s'))
{-7} data(linear_resolve(a', Read)(state($q'(s')$))) =
data(linear_resolve(a', Read)(s'))
{-8} transformer_invariant?(pm′‘states, transformers′)
{-9} transformers′(q')
{-10} pm′‘states = pm_phy‘states
{-11} ∀ (x: Memory_Address_4G):
addresses′(x) ⇒
restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))(x)
{-12} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a')
{-13} OK?($q'(s')$)
{-14} in_memory(min_linear, max_linear)(a')
{-15} Mem?(type_of(a'))
{-16} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)(s')))
(state(linear_resolve(a', Read)(s'))))
{-17} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a', Read)(state($q'(s')$))))
(state(linear_resolve(a', Read)(state($q'(s')$)))))

{1} union[Memory_access]
(singleton[Memory_access](Read), singleton[Memory_access](Execute))(Read)

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.2.1.2.1`.

`linear_unchanged_invariant.1.1.1.2.1.2.2:`

```
{-1}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states(state(q'(s')))
{-4}  pm'`states(s')
{-5}  OK?(linear_resolve(a', Read)(state(q'(s'))))
{-6}  OK?(linear_resolve(a', Read)(s'))
{-7}  data(linear_resolve(a', Read)(state(q'(s')))) =
        data(linear_resolve(a', Read)(s'))
{-8}  transformer_invariant?(pm'`states, transformers')
{-9}  transformers'(q')
{-10} pm'`states = pm_phy`states
{-11} ∀ (x: Memory_Address_4G):
        addresses'(x) ⇒
          restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr))(x)
{-12} extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-13} OK?(q'(s'))
{-14} in_memory(min_linear, max_linear)(a')
{-15} Mem?(type_of(a'))
{-16} OK?(memory_read(pm_phy`mem)
                     (data(linear_resolve(a', Read)(s')))
                     (state(linear_resolve(a', Read)(s'))))
{-17} OK?(memory_read(pm_phy`mem)
                     (data(linear_resolve(a', Read)(state(q'(s')))))
                     (state(linear_resolve(a', Read)(state(q'(s'))))))
-------------------------------------------------------------------------
 {1}  union[Memory_Address_4G]
        (restrict[Address, Memory_Address_4G, boolean](pm'`ro_addr),
          restrict[Address, Memory_Address_4G, boolean](pm'`rw_addr))
        (a')
```

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.1.2.1.2.2`.

`linear_unchanged_invariant.1.1.1.2.2:`

{-1} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{-2} is_linear_plain_memory?(pm′)
{-3} pm′‘states(state(q′(s′)))
{-4} pm′‘states(s′)
{-5} OK?(linear_resolve(a′, Read)(state(q′(s′))))
{-6} OK?(linear_resolve(a′, Read)(s′))
{-7} data(linear_resolve(a′, Read)(state(q′(s′)))) =
data(linear_resolve(a′, Read)(s′))
{-8} transformer_invariant?(pm′‘states, transformers′)
{-9} transformers′(q′)
{-10} pm′‘states = pm_phy‘states
{-11} ∀ (x: Memory_Address_4G):
(x ∈ addresses′) ⇒
(x ∈ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-12} extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a′)
{-13} OK?(q′(s′))
{-14} in_memory(min_linear, max_linear)(a′)
{-15} Mem?(type_of(a′))
{-16} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(s′)))
(state(linear_resolve(a′, Read)(s′))))
{-17} OK?(memory_read(pm_phy‘mem)
(data(linear_resolve(a′, Read)(state(q′(s′)))))
(state(linear_resolve(a′, Read)(state(q′(s′))))))

{1} Mem?(data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a′, Read)(s′))‘type_of)
∧
0 ≤
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a′, Read)(s′))‘offset
∧
data[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a′, Read)(s′))‘offset
< max_linear_offset
{2} union(pm_phy‘ro_addr, pm_phy‘rw_addr)(data(linear_resolve(a′, Read)(s′)))

Using lemma linear_resolve_memory_address,

Expanding the definition of every,

which is trivially true.

This completes the proof of `linear_unchanged_invariant.1.1.1.2.2`.

`linear_unchanged_invariant.1.1.2:`

```
{-1}  union(pm'`ro_addr, pm'`rw_addr)(a')
{-2}  transformer_invariant?(pm'`states, transformers')
{-3}  is_linear_plain_memory?(pm')
{-4}  pm'`states = pm_phy`states
{-5}  (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
{-6}  ∀ (s: (pm'`states)):
        ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
             a: Address):
          pm_phy`states(s_1) ∧
           transformers'(q) ∧
            extend[Address, Memory_Address_4G, bool, FALSE]
                 (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
                 (a)
              ∧
              OK?(q(s_1)) ∧
               OK?(memory_read(pm_phy`mem)(a)(s_1)) ∧
                OK?(memory_read(pm_phy`mem)(a)(state(q(s_1))))
            ⊃
            data(memory_read(pm_phy`mem)(a)(state(q(s_1)))) =
             data(memory_read(pm_phy`mem)(a)(s_1))
{-7}  pm'`states(s')
{-8}  transformers'(q')
{-9}  extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-10} OK?(q'(s'))
{-11} in_memory(min_linear, max_linear)(a')
{-12} OK?(memory_read(pm_phy`mem)(a')(s'))
{-13} OK?(memory_read(pm_phy`mem)(a')(state(q'(s'))))
-------------------------------------------------------------------------------
 {1}  Mem?(type_of(a'))
 {2}  data(memory_read(pm_phy`mem)(a')(state(q'(s')))) =
       data(memory_read(pm_phy`mem)(a')(s'))
```

Using lemma pm_memory_addr,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `linear_unchanged_invariant.1.1.2`.

linear_unchanged_invariant.1.2:

```
{-1}  transformer_invariant?(pm'`states, transformers')
{-2}  is_linear_plain_memory?(pm')
{-3}  pm'`states = pm_phy`states
{-4}  (addresses' ⊆ restrict[Address, Memory_Address_4G, boolean]((pm'`ro_addr ∪ pm'`rw_addr)))
{-5}  ∀ (s: (pm'`states)):
        ∀ (s_1: Physical_memory, q: [Physical_memory → SuperResult[Physical_memory]],
           a: Address):
          pm_phy`states(s_1) ∧
           transformers'(q) ∧
            extend[Address, Memory_Address_4G, bool, FALSE]
                (virt_to_phys_range(s, addresses', (singleton(Read) ∪ singleton(Execute))))
                (a)
             ∧
             OK?(q(s_1)) ∧
              OK?(memory_read(pm_phy`mem)(a)(s_1)) ∧
               OK?(memory_read(pm_phy`mem)(a)(state(q(s_1))))
          ⊃
          data(memory_read(pm_phy`mem)(a)(state(q(s_1)))) =
           data(memory_read(pm_phy`mem)(a)(s_1))
{-6}  pm'`states(s')
{-7}  transformers'(q')
{-8}  extend[Address, Memory_Address_4G, bool, FALSE](addresses')(a')
{-9}  OK?(q'(s'))
{-10} OK?(IF in_memory(min_linear, max_linear)(a')
            THEN IF Mem?(type_of(a'))
                   THEN (linear_resolve(a', Read) ##
                           (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                            (s')
                   ELSE memory_read(pm_phy`mem)(a')(s')
                   ENDIF
            ELSE fatal_result(s')
            ENDIF)
{-11} OK?(IF in_memory(min_linear, max_linear)(a')
            THEN IF Mem?(type_of(a'))
                   THEN (linear_resolve(a', Read) ##
                           (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                            (state(q'(s')))
                   ELSE memory_read(pm_phy`mem)(a')(state(q'(s')))
                   ENDIF
            ELSE fatal_result(state(q'(s')))
            ENDIF)
-------------------------------------------------------------------------------
{1}   union(pm'`ro_addr, pm'`rw_addr)(a')
{2}   IF in_memory(min_linear, max_linear)(a')
        THEN IF Mem?(type_of(a'))
               THEN data((linear_resolve(a', Read) ##
                           (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                            (state(q'(s'))))
               ELSE data(memory_read(pm_phy`mem)(a')(state(q'(s'))))
               ENDIF
      ELSE data(fatal_result(state(q'(s'))))
      ENDIF
       =
       IF in_memory(min_linear, max_linear)(a')
         THEN IF Mem?(type_of(a'))
                THEN data((linear_resolve(a', Read) ##
                           (λ (pa: Address): memory_read(pm_phy`mem)(pa)))
                            (s'))
                ELSE data(memory_read(pm_phy`mem)(a')(s'))
                ENDIF
       ELSE data(fatal_result(s'))
       ENDIF
```

Keeping (-4 -8 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `linear_unchanged_invariant.1.2`.
`linear_unchanged_invariant.2:`

```
{-1}  is_linear_plain_memory?(pm′) ⊃ pm′‘states = pm_phy‘states
{-2}  is_linear_plain_memory?(pm′)
{-3}  (addresses′ ⊆ restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))
{-4}  ∀ (s: (pm′‘states)):
        transformer_invariant?(pm_phy‘states, transformers′) ∧
         (∀ (s_1: Physical_memory, q: [Physical_memory → SuperRe-
      sult[Physical_memory]],
                a: Address):
              pm_phy‘states(s_1) ∧
               transformers′(q) ∧
                extend[Address, Memory_Address_4G, bool, FALSE]
                     (virt_to_phys_range(s, addresses′,
                                         (singleton(Read) ∪ singleton(Execute))))
                     (a)
                ∧
                OK?(q(s_1)) ∧
                 OK?(memory_read(pm_phy‘mem)(a)(s_1)) ∧
                  OK?(memory_read(pm_phy‘mem)(a)(state(q(s_1))))
              ⊃
              data(memory_read(pm_phy‘mem)(a)(state(q(s_1)))) =
               data(memory_read(pm_phy‘mem)(a)(s_1)))
-------------------------------------------------------------------------------
{1}   transformer_invariant?(pm′‘states, transformers′)
{2}   transformer_invariant?(pm′‘states, transformers′) ∧
       (∀ (s: Linear_memory[Physical_memory, pm_phy],
             q: [Linear_memory[Physical_memory, pm_phy] → SuperRe-
      sult[Physical_memory]],
             a: Address):
           pm′‘states(s) ∧
            transformers′(q) ∧
             extend[Address, Memory_Address_4G, bool, FALSE](addresses′)(a) ∧
              OK?(q(s)) ∧
               OK?(memory_read(pm′‘mem)(a)(s)) ∧
                OK?(memory_read(pm′‘mem)(a)(state(q(s))))
           ⊃
           data(memory_read(pm′‘mem)(a)(state(q(s)))) =
            data(memory_read(pm′‘mem)(a)(s)))
```

Hiding formulas: 2,
Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `linear_unchanged_invariant.2`.
Q.E.D.

C.117.85 Linear_Memory_Properties.pm_phy_read_after_resolve_ok_TCC1

Terse proof for `pm_phy_read_after_resolve_ok_TCC1`.

`pm_phy_read_after_resolve_ok_TCC1`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], a_2: Memory_Address_4G, ac: Memory_access):
plain_memory?(pm_phy) ∧ OK?(linear_resolve(a_2, ac)(s)) ⊃
OK?[Physical_memory, Address](linear_resolve[Physical_memory, pm_phy](a_2, ac)(s)) ∨
Exception?[Physical_memory, Address]
(linear_resolve[Physical_memory, pm_phy](a_2, ac)(s))

Repeatedly Skolemizing and flattening,

This completes the proof of `pm_phy_read_after_resolve_ok_TCC1`.

Q.E.D.

C.117.86 Linear_Memory_Properties.pm_phy_read_after_resolve_ok

Terse proof for `pm_phy_read_after_resolve_ok`.

`pm_phy_read_after_resolve_ok`:

{1} ∀ (s: Linear_memory[Physical_memory, pm_phy], a_1: Address, a_2: Memory_Address_4G,
ac: Memory_access):
OK?(linear_resolve(a_2, ac)(s)) ∧
plain_memory?(pm_phy) ∧
pm_phy`states(state(linear_resolve(a_2, ac)(s))) ∧
union(pm_phy`ro_addr, pm_phy`rw_addr)(a_1)
⊃ OK?(memory_read(pm_phy`mem)(a_1)(state(linear_resolve(a_2, ac)(s))))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_transformers_ok_read_ro_rw,

Using lemma expr_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of memory_read_transformers,

Instantiating quantified variables,

This completes the proof of `pm_phy_read_after_resolve_ok`.

Q.E.D.

C.117.87 Linear_Memory_Properties.pm_resolve_address

Terse proof for `pm_resolve_address`.

`pm_resolve_address:`

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], a: Memory_Address_4G):
is_linear_plain_memory?(pm) ∧
pm‘states(s) ∧
virt_to_phys_range(s, restrict[Address, Memory_Address_4G, boolean]((pm‘ro_addr ∪ pm‘rw_addr)), (singleton(Read) ∪ singleton(Execute)))(a)
⊃
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), extend[Address, Memory_Address_4G, bool, FALSE](address_in_pt_range?(s, restrict[Address, Memory_Address_4G, boolean]((pm‘ro_addr ∪ pm‘rw_addr)))))(a)

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 -3 -4 -6 -7 -9),
Case splitting on union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)),
we get 2 subgoals:
`pm_resolve_address.1:`

{-1} (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Ad
= restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} (virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict
{-3} disjoint?(virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_a
address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_
{-4} Mem?(a'‘type_of)
{-5} 0 ≤ a'‘offset
{-6} a'‘offset < max_linear_offset
{-7} virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)), (singleton(Read) ∪ singleton(Execute)))(a')

{1} difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), extend[Address, Memory_Address_4G, bool, FALSE](address_in_pt_range?(s', restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))))(a')

Replacing using formula -1,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
we get 2 subgoals:
`pm_resolve_address.1.1:`

{-1} pm_phy‘ro_addr(a')
{-2} Mem?(a'‘type_of)
{-3} $0 \leq a'$‘offset
{-4} a'‘offset $<$ expt(2, bus_width)
{-5} virt_to_phys_range(s',
restrict[Address, Memory_Address_4G, boolean]
((pm$'$‘ro_addr $\cup$ pm$'$‘rw_addr)),
(singleton(Read) $\cup$ singleton(Execute)))
(a')
{-6} address_in_pt_range?(s',
restrict[Address, Memory_Address_4G, boolean]
((pm$'$‘ro_addr $\cup$ pm$'$‘rw_addr)))
(a')

{1} virt_to_phys_range(s',
restrict[Address, Memory_Address_4G, boolean]
((pm$'$‘ro_addr $\cup$ pm$'$‘rw_addr)))
(a')

Keeping (-5 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_resolve_address.1.1`.
`pm_resolve_address.1.2:`

{-1} pm_phy‘rw_addr(a')
{-2} Mem?(a'‘type_of)
{-3} $0 \leq a'$‘offset
{-4} a'‘offset $<$ expt(2, bus_width)
{-5} virt_to_phys_range(s',
restrict[Address, Memory_Address_4G, boolean]
((pm$'$‘ro_addr $\cup$ pm$'$‘rw_addr)),
(singleton(Read) $\cup$ singleton(Execute)))
(a')
{-6} address_in_pt_range?(s',
restrict[Address, Memory_Address_4G, boolean]
((pm$'$‘ro_addr $\cup$ pm$'$‘rw_addr)))
(a')

{1} virt_to_phys_range(s',
restrict[Address, Memory_Address_4G, boolean]
((pm$'$‘ro_addr $\cup$ pm$'$‘rw_addr)))
(a')

Keeping (-5 1) and hiding *,
Expanding the definition of virt_to_phys_range,
Expanding the definition of virt_to_phys_range,
Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pm_resolve_address.1.2`.

`pm_resolve_address.2`:

{-1} (virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict
{-2} disjoint?(virt_to_phys_range(s',
(restrict [Address, Memory_Address_4G, boolean](pm'ro_ad
address_in_pt_range?(s',
(restrict [Address, Memory_Address_4G, boolean](pm'ro.
{-3} Mem?(a''type_of)
{-4} $0 \leq a'$'offset
{-5} a''offset $<$ max_linear_offset
{-6} virt_to_phys_range(s',
restrict[Address, Memory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)),
(singleton(Read) ∪ singleton(Execute)))
(a')

{1} (restrict[Address, Memory_Address_4G, boolean](pm'ro_addr) ∪ restrict [Address, Memory_Add
= restrict[Address, Memory_Address_4G, boolean]((pm'ro_addr ∪ pm'rw_addr))
{2} difference((pm_phy'ro_addr ∪ pm_phy'rw_addr),
extend[Address, Memory_Address_4G, bool, FALSE]
(address_in_pt_range?(s',
restrict[Address, Mem-
ory_Address_4G, boolean]
((pm'ro_addr ∪ pm'rw_addr)))))
(a')

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_resolve_address.2`.

Q.E.D.

C.117.88 Linear_Memory_Properties.pm_resolve_address_write

Terse proof for `pm_resolve_address_write`.

pm_resolve_address_write:

{1} ∀ (pm: Plain_Memory[Linear_memory], s: Linear_memory[Physical_memory, pm_phy], a: Memory_Address_4G):
is_linear_plain_memory?(pm) ∧
pm‘states(s) ∧
virt_to_phys_range(s, restrict[Address, Memory_Address_4G, boolean](pm‘rw_addr), singleton(Write))
(a)
⊃
difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), extend[Address, Memory_Address_4G, bool, FALSE](address_in_pt_range?(s, restrict[Address, Memory_Address_4G, boolean]((pm‘ro_addr ∪ pm‘rw_addr)))))
(a)

Repeatedly Skolemizing and flattening,
Using lemma pm_linear_blessed,
Expanding the definition of linear_blessed?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 -2 -3 -4 -5 -7 -9),
Case splitting on union(restrict[Address, Memory_Address_4G, boolean](pm!1‘ro_addr), restrict[Address, Memory_Address_4G, boolean](pm!1‘rw_addr)) = restrict[Address, Memory_Address_4G, boolean] (union(pm!1‘ro_addr, pm!1‘rw_addr)),
we get 2 subgoals:

pm_resolve_address_write.1:

{-1} (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Address_4G, b = restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} (virt_to_phys_range(s', restrict [Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write)) ⊆
{-3} disjoint?(virt_to_phys_range(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restri address_in_pt_range?(s', (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ res
{-4} Mem?(a'‘type_of)
{-5} 0 ≤ a'‘offset
{-6} a'‘offset < max_linear_offset
{-7} virt_to_phys_range(s', restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write))
(a')
{1} difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr), extend[Address, Memory_Address_4G, bool, FALSE](address_in_pt_range?(s', restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr)))))
(a')

Replacing using formula -1,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,

Keeping (-5 1) and hiding *,

Expanding the definition of virt_to_phys_range,

Expanding the definition of virt_to_phys_range,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

we get 2 subgoals:

`pm_resolve_address_write.1.1:`

{-1}	Mem?(lin_a′‘type_of)
{-2}	$0 \leq$ lin_a′‘offset
{-3}	lin_a′‘offset $<$ max_linear_offset
{-4}	restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(lin_a′)
{-5}	singleton(Write)(ac′)
{-6}	OK?(linear_resolve(lin_a′, ac′)(s'))
{-7}	data(linear_resolve(lin_a′, ac′)(s')) $=$ a'
{1}	OK?(linear_resolve(lin_a′, ac′)(s')) $\wedge$ data(linear_resolve(lin_a′, ac′)(s')) $=$ a'

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pm_resolve_address_write.1.1`.

`pm_resolve_address_write.1.2:`

{-1}	Mem?(lin_a′‘type_of)
{-2}	$0 \leq$ lin_a′‘offset
{-3}	lin_a′‘offset $<$ max_linear_offset
{-4}	restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr)(lin_a′)
{-5}	singleton(Write)(ac′)
{-6}	OK?(linear_resolve(lin_a′, ac′)(s'))
{-7}	data(linear_resolve(lin_a′, ac′)(s')) $=$ a'
{1}	pm′‘ro_addr(lin_a′) $\vee$ pm′‘rw_addr(lin_a′)

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_resolve_address_write.1.2`.

`pm_resolve_address_write.2:`

```
{-1}  (virt_to_phys_range(s′, restrict [Address, Memory_Address_4G, boolean](pm′‘rw_addr), singleton(Write)) ⊆
{-2}  disjoint?(virt_to_phys_range(s′,
                                   (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restri
                address_in_pt_range?(s′,
                                     (restrict [Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ res
{-3}  Mem?(a′‘type_of)
{-4}  0 ≤ a′‘offset
{-5}  a′‘offset < max_linear_offset
{-6}  virt_to_phys_range(s′, restrict[Address, Memory_Address_4G, boolean](pm′‘rw_addr),
                           singleton(Write))
                         (a′)
------------------------------------------------------------------------------
{1}   (restrict[Address, Memory_Address_4G, boolean](pm′‘ro_addr) ∪ restrict [Address, Memory_Address_4G, b
       = restrict[Address, Memory_Address_4G, boolean]((pm′‘ro_addr ∪ pm′‘rw_addr))
{2}   difference((pm_phy‘ro_addr ∪ pm_phy‘rw_addr),
                 extend[Address, Memory_Address_4G, bool, FALSE]
                     (address_in_pt_range?(s′,
                                           restrict[Address, Mem-
ory_Address_4G, boolean]
                                                    ((pm′‘ro_addr ∪ pm′‘rw_addr)))))
                (a′)
```

Keeping (1) and hiding *,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_resolve_address_write.2`.
Q.E.D.

C.118 Proofs for List_Split (linear_memory.pvs)

C.118.1 List_Split.split_TCC1

Terse proof for `split_TCC1`.

`split_TCC1:`

```
------------------------------------------------------------------------------
{1}   ∀ (size: nat, a: Address, bl: list[Byte], e2size: posnat):
        ¬ a + length(bl) ≤ floor(e2size)(a) + e2size ∧
          ¬ null?(bl) ∧ e2size = expt(2, size)
          ⊃
          (∀ (δ: {r: mod(e2size) | ∃ q: offset(a) = r + e2size × q}):
               δ = rem(e2size)(offset(a)) ⊃
                 e2size − δ ≥ 0 ∧ e2size − δ ≤ length[Byte](bl))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_TCC1`.
Q.E.D.

C.118.2 List_Split.split_TCC2

Terse proof for `split_TCC2`.

`split_TCC2:`

```
{1}  ∀ (size: nat, a: Address, bl: list[Byte], e2size: posnat):
       ¬ a + length(bl) ≤ floor(e2size)(a) + e2size ∧
         ¬ null?(bl) ∧ e2size = expt(2, size)
         ⊃
         (∀ (δ: {r: mod(e2size) | ∃ q: offset(a) = r + e2size × q}):
              δ = rem(e2size)(offset(a)) ⊃
                length[Byte](tail[Byte](bl, e2size − δ)) < length[Byte](bl))
```

Repeatedly Skolemizing and flattening,
Replacing using formula -8,
Rewriting using length_tail, matching in *,
we get 2 subgoals:
`split_TCC2.1:`

```
{-1}  integer_pred(q′)
{-2}  rem(e2size′)(offset(a′)) < e2size′
{-3}  offset(a′) = rem(e2size′)(offset(a′)) + e2size′ × q′
{-4}  size′ ≥ 0
{-5}  every(λ (x: number):
                number_field_pred(x) ∧
                  real_pred(x) ∧
                    rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-6}  e2size′ > 0
{-7}  e2size′ = expt(2, size′)
{-8}  delta′ = rem(e2size′)(offset(a′))
------------------------------------------------------------------------
{1}   a′ + length(bl′) ≤ floor(e2size′)(a′) + e2size′
{2}   null?(bl′)
{3}   rem(e2size′)(offset(a′)) + length(bl′) − e2size′ < length[Byte](bl′)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_TCC2.1`.
`split_TCC2.2:`

```
{-1}  integer_pred(q′)
{-2}  rem(e2size′)(offset(a′)) < e2size′
{-3}  offset(a′) = rem(e2size′)(offset(a′)) + e2size′ × q′
{-4}  size′ ≥ 0
{-5}  every(λ (x: number):
                number_field_pred(x) ∧
                  real_pred(x) ∧
                    rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-6}  e2size′ > 0
{-7}  e2size′ = expt(2, size′)
{-8}  delta′ = rem(e2size′)(offset(a′))
------------------------------------------------------------------------
{1}   e2size′ − rem(e2size′)(offset(a′)) ≤ length(bl′)
{2}   a′ + length(bl′) ≤ floor(e2size′)(a′) + e2size′
{3}   null?(bl′)
{4}   length[Byte](tail[Byte](bl′, e2size′ − rem(e2size′)(offset(a′)))) <
        length[Byte](bl′)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_TCC2.2`.
Q.E.D.

C.118.3 List_Split.split_pair_cross_size

Terse proof for `split_pair_cross_size`.
`split_pair_cross_size:`

```
{1}  ∀ (size: nat, a1: Address, bl1: list[Byte], n: nat):
       LET e2size = expt(2, size) IN
         n = length(bl1) ⊃
          every(λ (a2: Address, bl2: list[Byte]):
                    rem(e2size)(offset(a2)) + length(bl2) ≤ e2size)
                (split(size, a1, bl1))
```

Inducting on n on formula 1 using induction scheme NAT_induction,
we get 2 subgoals:
`split_pair_cross_size.1:`

```
{-1}  ∀ (size: nat, a1: Address, bl1: list[Byte]):
        n' = length(bl1) ⊃
         every(λ (a2: Address, bl2: list[Byte]):
                   rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
               (split(size, a1, bl1))
-------------------------------------------------------------------------------
{1}  ∀ (size: nat, a1: Address, bl1: list[Byte]):
       LET e2size = expt(2, size) IN
         n' = length(bl1) ⊃
          every(λ (a2: Address, bl2: list[Byte]):
                    rem(e2size)(offset(a2)) + length(bl2) ≤ e2size)
                (split(size, a1, bl1))
```

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -3 with the terms: (size!1 a1!1 bl1!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_pair_cross_size.1`.
`split_pair_cross_size.2:`

```
{1}  ∀ j:
       (∀ k:
          k < j ⊃
           (∀ (size: nat, a1: Address, bl1: list[Byte]):
              k = length(bl1) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                         rem(expt(2, size))(offset(a2)) + length(bl2) ≤
                           expt(2, size))
                     (split(size, a1, bl1))))
        ⊃
        (∀ (size: nat, a1: Address, bl1: list[Byte]):
           j = length(bl1) ⊃
            every(λ (a2: Address, bl2: list[Byte]):
                      rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
                  (split(size, a1, bl1)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of split,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`split_pair_cross_size.2.1:`

{-1} $\text{size}' \geq 0$
{-2} every(λ (x: number):
number_field_pred(x) $\wedge$
real_pred(x) $\wedge$
rational_pred(x) $\wedge$ integer_pred(x) $\wedge$ $x \geq 0$ $\wedge$ $x <$ max_byte)
(bl1′)
{-3} $j' \geq 0$
{-4} $\forall$ k:
$k < j' \supset$
($\forall$ (size: nat, a_1: Address, bl1: list[Byte]):
k = length(bl1) $\supset$
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, size))(offset(a_2)) + length(bl2) $\leq$ expt(2, size))
(split(size, a_1, bl1)))
{-5} j' = length(bl1′)
{-6} null?(bl1′)

{1} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, size′))(offset(a_2)) + length(bl2) $\leq$ expt(2, size′))
(null)

Expanding the definition of every,
which is trivially true.
This completes the proof of `split_pair_cross_size.2.1`.
`split_pair_cross_size.2.2:`

{-1} $\text{size}' \geq 0$
{-2} every(λ (x: number):
number_field_pred(x) $\wedge$
real_pred(x) $\wedge$
rational_pred(x) $\wedge$ integer_pred(x) $\wedge$ $x \geq 0$ $\wedge$ $x <$ max_byte)
(bl1′)
{-3} $j' \geq 0$
{-4} $\forall$ k:
$k < j' \supset$
($\forall$ (size: nat, a_1: Address, bl1: list[Byte]):
k = length(bl1) $\supset$
every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, size))(offset(a_2)) + length(bl2) $\leq$ expt(2, size))
(split(size, a_1, bl1)))
{-5} j' = length(bl1′)
{-6} a_1' + length(bl1′) $\leq$ floor(expt(2, size′))(a_1') + expt(2, size′)

{1} null?(bl1′)
{2} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, size′))(offset(a_2)) + length(bl2) $\leq$ expt(2, size′))
(cons((a_1', bl1′), null))

Expanding the definition of every,
Expanding the definition of every,
Keeping (-6 2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_pair_cross_size.2.2`.
`split_pair_cross_size.2.3`:

```
{-1}  size' >= 0
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x >= 0 ∧ x < max_byte)
            (bl1')
{-3}  j' >= 0
{-4}  ∀ k:
        k < j' ⊃
         (∀ (size: nat, a1: Address, bl1: list[Byte]):
              k = length(bl1) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                         rem(expt(2, size))(offset(a2)) + length(bl2) <= expt(2, size))
                     (split(size, a1, bl1)))
{-5}  j' = length(bl1')
------------------------------------------------------------------------
{1}   null?(bl1')
{2}   a1' + length(bl1') <= floor(expt(2, size'))(a1') + expt(2, size')
{3}   every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, size'))(offset(a2)) + length(bl2) <= expt(2, size'))
            (cons((a1',
                   head(bl1',
                        expt(2, size') - rem(expt(2, size'))(offset(a1')))),
                  split(size', floor(expt(2, size'))(a1') + expt(2, size'),
                        tail(bl1',
                             expt(2, size') - rem(expt(2, size'))(offset(a1'))))))
```

Expanding the definition of every,
Applying propositional simplification,
we get 2 subgoals:
`split_pair_cross_size.2.3.1`:

```
{-1}  size' >= 0
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x >= 0 ∧ x < max_byte)
            (bl1')
{-3}  j' >= 0
{-4}  ∀ k:
        k < j' ⊃
         (∀ (size: nat, a1: Address, bl1: list[Byte]):
              k = length(bl1) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                         rem(expt(2, size))(offset(a2)) + length(bl2) <= expt(2, size))
                     (split(size, a1, bl1)))
{-5}  j' = length(bl1')
------------------------------------------------------------------------
{1}   rem(expt(2, size'))(offset(a1')) + length(head(bl1', expt(2, size') - rem(expt(2, size'))(offset(a1'))))
       <= expt(2, size')
{2}   null?(bl1')
{3}   a1' + length(bl1') <= floor(expt(2, size'))(a1') + expt(2, size')
```

Rewriting using length_head, matching in *,
we get 2 subgoals:
`split_pair_cross_size.2.3.1.1:`

```
{-1}  size′ ≥ 0
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl1′)
{-3}  j′ ≥ 0
{-4}  ∀ k:
        k < j′ ⊃
         (∀ (size: nat, a1: Address, bl1: list[Byte]):
              k = length(bl1) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                           rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
                     (split(size, a1, bl1)))
{-5}  j′ = length(bl1′)
──────────────────────────────────────────────────────────────────────────────
{1}   rem(expt(2, size′))(offset(a1′)) + expt(2, size′) − rem(expt(2, size′))(offset(a1′))
       ≤ expt(2, size′)
{2}   null?(bl1′)
{3}   a1′ + length(bl1′) ≤ floor(expt(2, size′))(a1′) + expt(2, size′)
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `split_pair_cross_size.2.3.1.1`.
`split_pair_cross_size.2.3.1.2:`

```
{-1}  size′ ≥ 0
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl1′)
{-3}  j′ ≥ 0
{-4}  ∀ k:
        k < j′ ⊃
         (∀ (size: nat, a1: Address, bl1: list[Byte]):
              k = length(bl1) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                           rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
                     (split(size, a1, bl1)))
{-5}  j′ = length(bl1′)
──────────────────────────────────────────────────────────────────────────────
{1}   expt(2, size′) − rem(expt(2, size′))(offset(a1′)) ≤ length(bl1′)
{2}   rem(expt(2, size′))(offset(a1′)) + length(head(bl1′, expt(2, size′) − rem(expt(2, size′))(offset(a1′))))
       ≤ expt(2, size′)
{3}   null?(bl1′)
{4}   a1′ + length(bl1′) ≤ floor(expt(2, size′))(a1′) + expt(2, size′)
```

Expanding the definition of floor,
Expanding the definition of floor,
Expanding the definition of <=,
Expanding the definition of +,
which is trivially true.
This completes the proof of `split_pair_cross_size.2.3.1.2`.

split_pair_cross_size.2.3.2:

```
{-1}  size′ ≥ 0
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl1′)
{-3}  j′ ≥ 0
{-4}  ∀ k:
        k < j′ ⊃
         (∀ (size: nat, a1: Address, bl1: list[Byte]):
              k = length(bl1) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                         rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
                     (split(size, a1, bl1)))
{-5}  j′ = length(bl1′)
-------------------------------------------------------------------------------
{1}   every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, size′))(offset(a2)) + length(bl2) ≤ expt(2, size′))
            (split(size′, floor(expt(2, size′))(a1′) + expt(2, size′),
                     tail(bl1′,
                          expt(2, size′) − rem(expt(2, size′))(offset(a1′)))))
{2}   null?(bl1′)
{3}   a1′ + length(bl1′) ≤ floor(expt(2, size′))(a1′) + expt(2, size′)
```

Instantiating the top quantifier in -4 with the terms: (length(tail(bl1!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a1!1))))),
Splitting conjunctions,
we get 2 subgoals:

split_pair_cross_size.2.3.2.1:

```
{-1}  ∀ (size: nat, a1: Address, bl1: list[Byte]):
        length(tail(bl1′, expt(2, size′) − rem(expt(2, size′))(offset(a1′)))) =
         length(bl1)
         ⊃
         every(λ (a2: Address, bl2: list[Byte]):
                   rem(expt(2, size))(offset(a2)) + length(bl2) ≤ expt(2, size))
               (split(size, a1, bl1))
{-2}  size′ ≥ 0
{-3}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl1′)
{-4}  j′ ≥ 0
{-5}  j′ = length(bl1′)
-------------------------------------------------------------------------------
{1}   every(λ (a2: Address, bl2: list[Byte]):
                rem(expt(2, size′))(offset(a2)) + length(bl2) ≤ expt(2, size′))
            (split(size′, floor(expt(2, size′))(a1′) + expt(2, size′),
                     tail(bl1′,
                          expt(2, size′) − rem(expt(2, size′))(offset(a1′)))))
{2}   null?(bl1′)
{3}   a1′ + length(bl1′) ≤ floor(expt(2, size′))(a1′) + expt(2, size′)
```

Instantiating the top quantifier in -1 with the terms: (size!1 floor(expt(2, size!1))(a1!1) + expt(2,

size!1) tail(bl1!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a1!1)))),

which is trivially true.

This completes the proof of `split_pair_cross_size.2.3.2.1`.

`split_pair_cross_size.2.3.2.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{size}' \geq 0 \\
\{\text{-}2\} & \text{every}(\lambda\ (x\text{: number}): \\
 & \qquad \text{number\_field\_pred}(x) \wedge \\
 & \qquad\ \text{real\_pred}(x) \wedge \\
 & \qquad\quad \text{rational\_pred}(x) \wedge \text{integer\_pred}(x) \wedge x \geq 0 \wedge x < \text{max\_byte}) \\
 & \quad (\text{bl1}') \\
\{\text{-}3\} & j' \geq 0 \\
\{\text{-}4\} & j' = \text{length}(\text{bl1}') \\
\hline
\{1\} & \text{length}(\text{tail}(\text{bl1}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')))) < \\
 & \ j' \\
\{2\} & \text{every}(\lambda\ (a_2\text{: Address}, \text{bl2: list}[\text{Byte}]): \\
 & \qquad \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{size}')) \\
 & \quad (\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}'), \\
 & \qquad\quad \text{tail}(\text{bl1}', \\
 & \qquad\qquad \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1'))))) \\
\{3\} & \text{null?}(\text{bl1}') \\
\{4\} & a_1' + \text{length}(\text{bl1}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')
\end{array}
$$

Rewriting using length_tail, matching in *,

we get 2 subgoals:

`split_pair_cross_size.2.3.2.2.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & \text{size}' \geq 0 \\
\{\text{-}2\} & \text{every}(\lambda\ (x\text{: number}): \\
 & \qquad \text{number\_field\_pred}(x) \wedge \\
 & \qquad\ \text{real\_pred}(x) \wedge \\
 & \qquad\quad \text{rational\_pred}(x) \wedge \text{integer\_pred}(x) \wedge x \geq 0 \wedge x < \text{max\_byte}) \\
 & \quad (\text{bl1}') \\
\{\text{-}3\} & j' \geq 0 \\
\{\text{-}4\} & j' = \text{length}(\text{bl1}') \\
\hline
\{1\} & \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')) + \text{length}(\text{bl1}') - \text{expt}(2, \text{size}') < j' \\
\{2\} & \text{every}(\lambda\ (a_2\text{: Address}, \text{bl2: list}[\text{Byte}]): \\
 & \qquad \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_2)) + \text{length}(\text{bl2}) \leq \text{expt}(2, \text{size}')) \\
 & \quad (\text{split}(\text{size}', \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}'), \\
 & \qquad\quad \text{tail}(\text{bl1}', \\
 & \qquad\qquad \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1'))))) \\
\{3\} & \text{null?}(\text{bl1}') \\
\{4\} & a_1' + \text{length}(\text{bl1}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')
\end{array}
$$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_pair_cross_size.2.3.2.2.1`.

`split_pair_cross_size.2.3.2.2.2:`

{-1} $\text{size}' \geq 0$
{-2} every(λ (x: number):
number_field_pred(x) $\wedge$
real_pred(x) $\wedge$
rational_pred(x) $\wedge$ integer_pred(x) $\wedge$ $x \geq 0$ $\wedge$ $x <$ max_byte)
(bl1$'$)
{-3} $j' \geq 0$
{-4} $j' = \text{length}(\text{bl1}')$

{1} $\text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')) \leq \text{length}(\text{bl1}')$
{2} $\text{length}(\text{tail}(\text{bl1}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a_1')))) <$
j'
{3} every(λ (a_2: Address, bl2: list[Byte]):
rem(expt(2, size$'$))(offset(a_2)) + length(bl2) $\leq$ expt(2, size$'$))
(split(size$'$, floor(expt(2, size$'$))(a_1') + expt(2, size$'$),
tail(bl1$'$,
expt(2, size$'$) − rem(expt(2, size$'$))(offset(a_1')))))
{4} null?(bl1$'$)
{5} $a_1' + \text{length}(\text{bl1}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a_1') + \text{expt}(2, \text{size}')$

Hiding formulas: -2, 3,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_pair_cross_size.2.3.2.2.2`.

Q.E.D.

C.118.4 List_Split.split_pair_concat

Terse proof for `split_pair_concat`.

`split_pair_concat:`

{1} $\forall$ (size: nat, a: Address, bl: list[Byte], n: nat):
LET e2size = expt(2, size) IN
n = length(bl) $\supset$
reduce(null,
λ (e: [Address, list[Byte]], tail: list[Byte]):
LET head = e‘2 IN append(head, tail))
(split(size, a, bl))
= bl

Inducting on n on formula 1 using induction scheme NAT_induction,

we get 2 subgoals:

`split_pair_concat.1:`

```
{-1}  ∀ (size: nat, a: Address, bl: list[Byte]):
        n′ = length(bl) ⊃
         reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
                (split(size, a, bl))
          = bl
------------------------------------------------------------------------------
{1}   ∀ (size: nat, a: Address, bl: list[Byte]):
        LET e2size = expt(2, size) IN
          n′ = length(bl) ⊃
           reduce(null,
                  λ (e: [Address, list[Byte]], tail: list[Byte]):
                    LET head = e‘2 IN append(head, tail))
                 (split(size, a, bl))
            = bl
```

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -2,

Instantiating quantified variables,

This completes the proof of `split_pair_concat.1`.

`split_pair_concat.2:`

```

------------------------------------------------------------------------------
{1}   ∀ j:
        (∀ k:
           k < j ⊃
            (∀ (size: nat, a: Address, bl: list[Byte]):
               k = length(bl) ⊃
                reduce(null,
                       λ (e: [Address, list[Byte]], tail: list[Byte]):
                         append(e‘2, tail))
                      (split(size, a, bl))
                 = bl))
         ⊃
         (∀ (size: nat, a: Address, bl: list[Byte]):
            j = length(bl) ⊃
             reduce(null,
                    λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
                   (split(size, a, bl))
              = bl)
```

Repeatedly Skolemizing and flattening,

Expanding the definition of split,

Expanding the definition of reduce,

Hiding formulas: -2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`split_pair_concat.2.1:`

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  ∀ k:
        k < j′ ⊃
         (∀ (size: nat, a: Address, bl: list[Byte]):
              k = length(bl) ⊃
               reduce(null,
                        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                      (split(size, a, bl))
                = bl)
{-4}  j′ = length(bl′)
{-5}  a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
-------------------------------------------------------------------------------
{1}   null?(bl′)
{2}   append(bl′,
               reduce(null,
                        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                      (null))
       = bl′
```

Expanding the definition of reduce,

Rewriting using append_null, matching in *,

This completes the proof of `split_pair_concat.2.1`.

`split_pair_concat.2.2:`

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  ∀ k:
        k < j′ ⊃
         (∀ (size: nat, a: Address, bl: list[Byte]):
              k = length(bl) ⊃
               reduce(null,
                        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                      (split(size, a, bl))
                = bl)
{-4}  j′ = length(bl′)
-------------------------------------------------------------------------------
{1}   null?(bl′)
{2}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{3}   append(head(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′))),
               reduce(null,
                        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                      (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                               tail(bl′,
                                      expt(2, size′) − rem(expt(2, size′))(offset(a′))))))
       = bl′
```

Instantiating the top quantifier in -3 with the terms: (length(tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1))))),

Splitting conjunctions,

we get 2 subgoals:

`split_pair_concat.2.2.1:`

```
{-1}  ∀ (size: nat, a: Address, bl: list[Byte]):
        length(tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))) =
          length(bl)
          ⊃
          reduce(null, λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
                 (split(size, a, bl))
            = bl
{-2}  size′ ≥ 0
{-3}  j′ ≥ 0
{-4}  j′ = length(bl′)
------------------------------------------------------------------------
{1}   null?(bl′)
{2}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{3}   append(head(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′))),
             reduce(null,
                    λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
                    (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                           tail(bl′,
                                expt(2, size′) − rem(expt(2, size′))(offset(a′))))))
        = bl′
```

Instantiating the top quantifier in -1 with the terms: (size!1 floor(expt(2, size!1))(a!1) + expt(2, size!1) tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1)))),

Replacing using formula -1,

Rewriting using head_tail, matching in *,

Expanding the definition of floor,

Expanding the definition of floor,

Expanding the definition of +,

Expanding the definition of <=,

which is trivially true.

This completes the proof of `split_pair_concat.2.2.1`.

`split_pair_concat.2.2.2:`

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  j′ = length(bl′)
------------------------------------------------------------------------
{1}   length(tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))) < j′
{2}   null?(bl′)
{3}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{4}   append(head(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′))),
             reduce(null,
                    λ (e: [Address, list[Byte]], tail: list[Byte]): append(e‘2, tail))
                    (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                           tail(bl′,
                                expt(2, size′) − rem(expt(2, size′))(offset(a′))))))
        = bl′
```

Rewriting using length_tail, matching in *,

we get 2 subgoals:

`split_pair_concat.2.2.2.1:`

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  j′ = length(bl′)
------------------------------------------------------------------------
{1}   rem(expt(2, size′))(offset(a′)) + length[Byte](bl′) − expt(2, size′) <
        j′
{2}   null?(bl′)
{3}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{4}   append(head(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′))),
               reduce(null,
                        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                      (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                                 tail(bl′,
                                        expt(2, size′) − rem(expt(2, size′))(offset(a′))))))
        = bl′
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `split_pair_concat.2.2.2.1`.
`split_pair_concat.2.2.2.2:`

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  j′ = length(bl′)
------------------------------------------------------------------------
{1}   expt(2, size′) − rem(expt(2, size′))(offset(a′)) ≤ length[Byte](bl′)
{2}   length(tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))) < j′
{3}   null?(bl′)
{4}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{5}   append(head(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′))),
               reduce(null,
                        λ (e: [Address, list[Byte]], tail: list[Byte]): append(e`2, tail))
                      (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                                 tail(bl′,
                                        expt(2, size′) − rem(expt(2, size′))(offset(a′))))))
        = bl′
```

Hiding formulas: 2, 5,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_pair_concat.2.2.2.2`.
Q.E.D.

C.118.5 List_Split.split_range

Terse proof for `split_range`.
`split_range:`

```
------------------------------------------------------------------------
{1}   ∀ (size: nat, a: Address, bl: list[Byte], n: nat):
        LET e2size = expt(2, size) IN
          n = length(bl) ⊃
            every(λ (a2: Address, bl2: list[Byte]):
                      a ≤ a2 ∧ a2 + length(bl2) ≤ a + n)
                  (split(size, a, bl))
```

Inducting on n on formula 1 using induction scheme NAT_induction,
we get 2 subgoals:

`split_range.1:`

```
{-1}  ∀ (size: nat, a: Address, bl: list[Byte]):
        n′ = length(bl) ⊃
         every(λ (a2: Address, bl2: list[Byte]):
                  a ≤ a2 ∧ a2 + length(bl2) ≤ a + n′)
               (split(size, a, bl))
------------------------------------------------------------
{1}   ∀ (size: nat, a: Address, bl: list[Byte]):
        LET e2size = expt(2, size) IN
          n′ = length(bl) ⊃
           every(λ (a2: Address, bl2: list[Byte]):
                    a ≤ a2 ∧ a2 + length(bl2) ≤ a + n′)
                 (split(size, a, bl))
```

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -3 with the terms: (size!1 a!1 bl!1),

which is trivially true.

This completes the proof of `split_range.1`.

`split_range.2:`

```
------------------------------------------------------------
{1}   ∀ j:
        (∀ k:
           k < j ⊃
            (∀ (size: nat, a: Address, bl: list[Byte]):
                 k = length(bl) ⊃
                  every(λ (a2: Address, bl2: list[Byte]):
                           a ≤ a2 ∧ a2 + length(bl2) ≤ a + k)
                        (split(size, a, bl))))
         ⊃
         (∀ (size: nat, a: Address, bl: list[Byte]):
              j = length(bl) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                        a ≤ a2 ∧ a2 + length(bl2) ≤ a + j)
                     (split(size, a, bl)))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: -2,

Expanding the definition of every,

Expanding the definition of split,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 6 subgoals:

`split_range.2.1:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall\ (\text{size}\colon \text{nat},\ a\colon \text{Address},\ \text{bl}\colon \text{list}[\text{Byte}])\colon$
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2\colon \text{Address},\ \text{bl2}\colon \text{list}[\text{Byte}])\colon$
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size},\ a,\ \text{bl})))$
{-4} $j' = \text{length}(\text{bl}')$
{-5} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$

{1} $\text{null?}(\text{bl}')$
{2} $\text{every}(\lambda\ (a_2\colon \text{Address},\ \text{bl2}\colon \text{list}[\text{Byte}])\colon$
$a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + j')$
(null)

Expanding the definition of every,
which is trivially true.
This completes the proof of `split_range.2.1`.

`split_range.2.2:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall\ (\text{size}\colon \text{nat},\ a\colon \text{Address},\ \text{bl}\colon \text{list}[\text{Byte}])\colon$
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2\colon \text{Address},\ \text{bl2}\colon \text{list}[\text{Byte}])\colon$
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size},\ a,\ \text{bl})))$
{-4} $j' = \text{length}(\text{bl}')$
{-5} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$

{1} $\text{null?}(\text{bl}')$
{2} $a' + \text{length}(\text{bl}') \leq a' + j'$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_range.2.2`.

`split_range.2.3:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall\ (\text{size}\colon \text{nat},\ a\colon \text{Address},\ \text{bl}\colon \text{list}[\text{Byte}])\colon$
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2\colon \text{Address},\ \text{bl2}\colon \text{list}[\text{Byte}])\colon$
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size},\ a,\ \text{bl})))$
{-4} $j' = \text{length}(\text{bl}')$
{-5} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$

{1} $\text{null?}(\text{bl}')$
{2} $a' \leq a'$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_range.2.3`.
`split_range.2.4`:

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  ∀ k:
        k < j′ ⊃
         (∀ (size: nat, a: Address, bl: list[Byte]):
              k = length(bl) ⊃
               every(λ (a2: Address, bl2: list[Byte]):
                         a ≤ a2 ∧ a2 + length(bl2) ≤ a + k)
                     (split(size, a, bl)))
{-4}  j′ = length(bl′)
------------------------------------------------------------------------
{1}   null?(bl′)
{2}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{3}   every(λ (a2: Address, bl2: list[Byte]):
                a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + j′)
            (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                          tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))))
```

Instantiating the top quantifier in -3 with the terms: (length(tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1))))),
Splitting conjunctions,
we get 2 subgoals:
`split_range.2.4.1`:

```
{-1}  ∀ (size: nat, a: Address, bl: list[Byte]):
        length(tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))) =
         length(bl)
         ⊃
         every(λ (a2: Address, bl2: list[Byte]):
                   a ≤ a2 ∧
                    a2 + length(bl2) ≤
                     a + length(tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))))
               (split(size, a, bl))
{-2}  size′ ≥ 0
{-3}  j′ ≥ 0
{-4}  j′ = length(bl′)
------------------------------------------------------------------------
{1}   null?(bl′)
{2}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{3}   every(λ (a2: Address, bl2: list[Byte]):
                a′ ≤ a2 ∧ a2 + length(bl2) ≤ a′ + j′)
            (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                          tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))))
```

Instantiating the top quantifier in -1 with the terms: (size!1 floor(expt(2, size!1))(a!1) + expt(2, size!1) tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1)))),
Using lemma every_implied,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: (-1 4),
Repeatedly Skolemizing and flattening,
Rewriting using length_tail, matching in *,
we get 2 subgoals:

`split_range.2.4.1.1:`

$$
\begin{array}{ll}
\{\text{-1}\} & \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}') \ \leq\ t'\text{`}1 \\
\{\text{-2}\} & t'\text{`}1 + \text{length}(t'\text{`}2) \ \leq \\
 & \quad \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}') + \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')) + \text{length}\big[\text{Byte}\big](\text{bl}') - \\
\{\text{-3}\} & \text{size}' \ \geq\ 0 \\
\{\text{-4}\} & j' \ \geq\ 0 \\
\{\text{-5}\} & j' \ =\ \text{length}(\text{bl}') \\
\hline
\{1\} & a' \ \leq\ t'\text{`}1 \ \wedge\ t'\text{`}1 + \text{length}(t'\text{`}2) \ \leq\ a' + j' \\
\{2\} & \text{null?}(\text{bl}') \\
\{3\} & a' + \text{length}(\text{bl}') \ \leq\ \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.4.1.1`.

`split_range.2.4.1.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}') \ \leq\ t'\text{`}1 \\
\{\text{-2}\} & t'\text{`}1 + \text{length}(t'\text{`}2) \ \leq \\
 & \quad \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}') + \text{length}(\text{tail}(\text{bl}',\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')))) \\
\{\text{-3}\} & \text{size}' \ \geq\ 0 \\
\{\text{-4}\} & j' \ \geq\ 0 \\
\{\text{-5}\} & j' \ =\ \text{length}(\text{bl}') \\
\hline
\{1\} & \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')) \ \leq\ \text{length}\big[\text{Byte}\big](\text{bl}') \\
\{2\} & a' \ \leq\ t'\text{`}1 \ \wedge\ t'\text{`}1 + \text{length}(t'\text{`}2) \ \leq\ a' + j' \\
\{3\} & \text{null?}(\text{bl}') \\
\{4\} & a' + \text{length}(\text{bl}') \ \leq\ \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.4.1.2`.

`split_range.2.4.2:`

$$
\begin{array}{ll}
\{\text{-1}\} & \text{size}' \ \geq\ 0 \\
\{\text{-2}\} & j' \ \geq\ 0 \\
\{\text{-3}\} & j' \ =\ \text{length}(\text{bl}') \\
\hline
\{1\} & \text{length}(\text{tail}(\text{bl}',\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')))) \ <\ j' \\
\{2\} & \text{null?}(\text{bl}') \\
\{3\} & a' + \text{length}(\text{bl}') \ \leq\ \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}') \\
\{4\} & \text{every}(\lambda\ (a_2\texttt{: Address, bl2: list}\big[\text{Byte}\big])\texttt{:} \\
 & \qquad a' \ \leq\ a_2 \ \wedge\ a_2 + \text{length}(\text{bl2}) \ \leq\ a' + j') \\
 & \qquad (\text{split}(\text{size}',\ \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}'), \\
 & \qquad\quad \text{tail}(\text{bl}',\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')))))
\end{array}
$$

Rewriting using length_tail, matching in *,

we get 2 subgoals:

`split_range.2.4.2.1:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $j' = \text{length}(\text{bl}')$

{1} $\text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')) + \text{length}\lceil\text{Byte}\rceil(\text{bl}') - \text{expt}(2,\ \text{size}') < j'$
{2} $\text{null?}(\text{bl}')$
{3} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$
{4} $\text{every}(\lambda\ (a_2\texttt{: Address, bl2: list}\lceil\text{Byte}\rceil)\texttt{:}$
$a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + j')$
$(\text{split}(\text{size}'\texttt{,}\ \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')\texttt{,}$
$\text{tail}(\text{bl}'\texttt{,}\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a'))))))$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `split_range.2.4.2.1`.

`split_range.2.4.2.2:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $j' = \text{length}(\text{bl}')$

{1} $\text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')) \leq \text{length}\lceil\text{Byte}\rceil(\text{bl}')$
{2} $\text{length}(\text{tail}(\text{bl}'\texttt{,}\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')))) < j'$
{3} $\text{null?}(\text{bl}')$
{4} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$
{5} $\text{every}(\lambda\ (a_2\texttt{: Address, bl2: list}\lceil\text{Byte}\rceil)\texttt{:}$
$a' \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a' + j')$
$(\text{split}(\text{size}'\texttt{,}\ \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')\texttt{,}$
$\text{tail}(\text{bl}'\texttt{,}\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a'))))))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.4.2.2`.

`split_range.2.5:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k\texttt{:}$
$k < j' \supset$
$(\forall\ (\text{size}\texttt{: nat, } a\texttt{: Address, bl: list}\lceil\text{Byte}\rceil)\texttt{:}$
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2\texttt{: Address, bl2: list}\lceil\text{Byte}\rceil)\texttt{:}$
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size}\texttt{, } a\texttt{, bl})))$
{-4} $j' = \text{length}(\text{bl}')$

{1} $\text{null?}(\text{bl}')$
{2} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$
{3} $a' + \text{length}(\text{head}(\text{bl}'\texttt{,}\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a'))))$
$\leq a' + j'$

Rewriting using length_head, matching in *,

we get 2 subgoals:

`split_range.2.5.1:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall$ (size: nat, a: Address, bl: list[Byte]):
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]):
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size}, a, \text{bl})))$
{-4} $j' = \text{length}(\text{bl}')$

{1} $\text{null?}(\text{bl}')$
{2} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}')$
{3} $a' + \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')) \leq a' + j'$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_range.2.5.1`.

`split_range.2.5.2:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall$ (size: nat, a: Address, bl: list[Byte]):
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]):
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size}, a, \text{bl})))$
{-4} $j' = \text{length}(\text{bl}')$

{1} $\text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a')) \leq \text{length}[\text{Byte}](\text{bl}')$
{2} $\text{null?}(\text{bl}')$
{3} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}')$
{4} $a' + \text{length}(\text{head}(\text{bl}', \text{expt}(2, \text{size}') - \text{rem}(\text{expt}(2, \text{size}'))(\text{offset}(a'))))$
$\leq a' + j'$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_range.2.5.2`.

`split_range.2.6:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall$ (size: nat, a: Address, bl: list[Byte]):
$k = \text{length}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]):
$a \leq a_2 \wedge a_2 + \text{length}(\text{bl2}) \leq a + k)$
$(\text{split}(\text{size}, a, \text{bl})))$
{-4} $j' = \text{length}(\text{bl}')$

{1} $\text{null?}(\text{bl}')$
{2} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2, \text{size}'))(a') + \text{expt}(2, \text{size}')$
{3} $a' \leq a'$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_range.2.6`.

Q.E.D.

C.118.6 List_Split.split_no_null

Terse proof for `split_no_null`.

`split_no_null`:

```
{1}  ∀ (size: nat, a: Address, bl: list[Byte], n: nat):
       LET e2size = expt(2, size) IN
         n = length(bl) ∧ cons?(bl) ⊃
           every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl))
```

Inducting on n on formula 1 using induction scheme NAT_induction,

we get 2 subgoals:

`split_no_null.1`:

```
{-1}  ∀ (size: nat, a: Address, bl: list[Byte]):
        n′ = length(bl) ∧ cons?(bl) ⊃
          every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl))
{1}   ∀ (size: nat, a: Address, bl: list[Byte]):
        LET e2size = expt(2, size) IN
          n′ = length(bl) ∧ cons?(bl) ⊃
            every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `split_no_null.1`.

`split_no_null.2`:

```
{1}  ∀ j:
       (∀ k:
          k < j ⊃
            (∀ (size: nat, a: Address, bl: list[Byte]):
               k = length(bl) ∧ cons?(bl) ⊃
                 every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))
                   (split(size, a, bl))))
        ⊃
        (∀ (size: nat, a: Address, bl: list[Byte]):
           j = length(bl) ∧ cons?(bl) ⊃
             every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl)))
```

Repeatedly Skolemizing and flattening,

Hiding formulas: -2,

Expanding the definition of every,

Expanding the definition of split,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`split_no_null.2.1:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall$ (size: nat, a: Address, bl: list[Byte]):
$k = \text{length}(\text{bl}) \wedge \text{cons?}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl)))
{-4} $j' = \text{length}(\text{bl}')$
{-5} $\text{cons?}(\text{bl}')$
{-6} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$

{1} $\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]): cons?(bl2))(null)

Expanding the definition of every,
which is trivially true.
This completes the proof of `split_no_null.2.1`.
`split_no_null.2.2:`

{-1} $\text{size}' \geq 0$
{-2} $j' \geq 0$
{-3} $\forall\ k$:
$k < j' \supset$
$(\forall$ (size: nat, a: Address, bl: list[Byte]):
$k = \text{length}(\text{bl}) \wedge \text{cons?}(\text{bl}) \supset$
$\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl)))
{-4} $j' = \text{length}(\text{bl}')$
{-5} $\text{cons?}(\text{bl}')$

{1} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$
{2} $\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]): cons?(bl2))
(split(size', floor(expt(2, size'))(a') + expt(2, size'),
tail(bl', expt(2, size') − rem(expt(2, size'))(offset(a')))))

Instantiating the top quantifier in -3 with the terms: (length(tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1))))),
Splitting conjunctions,
we get 2 subgoals:
`split_no_null.2.1.1:`

{-1} $\forall$ (size: nat, a: Address, bl: list[Byte]):
$\text{length}(\text{tail}(\text{bl}',\ \text{expt}(2,\ \text{size}') - \text{rem}(\text{expt}(2,\ \text{size}'))(\text{offset}(a')))) =$
length(bl)
$\wedge$ cons?(bl)
$\supset$ every($\lambda\ (a_2$: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl))
{-2} $\text{size}' \geq 0$
{-3} $j' \geq 0$
{-4} $j' = \text{length}(\text{bl}')$
{-5} $\text{cons?}(\text{bl}')$

{1} $a' + \text{length}(\text{bl}') \leq \text{floor}(\text{expt}(2,\ \text{size}'))(a') + \text{expt}(2,\ \text{size}')$
{2} $\text{every}(\lambda\ (a_2$: Address, bl2: list[Byte]): cons?(bl2))
(split(size', floor(expt(2, size'))(a') + expt(2, size'),
tail(bl', expt(2, size') − rem(expt(2, size'))(offset(a')))))

Instantiating the top quantifier in -1 with the terms: (size!1 floor(expt(2, size!1))(a!1) + expt(2, size!1) tail(bl!1, expt(2, size!1) - rem(expt(2, size!1))(offset(a!1)))),

Simplifying, rewriting, and recording with decision procedures,
Using lemma length_tail,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_no_null.2.2.1`.
`split_no_null.2.2.2`:

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  j′ = length(bl′)
{-4}  cons?(bl′)
------------------------------------------------------------------------
{1}   length(tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))) < j′
{2}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{3}   every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))
            (split(size′, floor(expt(2, size′))(a′) + expt(2, size′),
                          tail(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′)))))
```

Hiding formulas: 3,
Using lemma length_tail,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_no_null.2.2.2`.
`split_no_null.2.3`:

```
{-1}  size′ ≥ 0
{-2}  j′ ≥ 0
{-3}  ∀ k:
        k < j′ ⊃
          (∀ (size: nat, a: Address, bl: list[Byte]):
              k = length(bl) ∧ cons?(bl) ⊃
                every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(size, a, bl)))
{-4}  j′ = length(bl′)
{-5}  cons?(bl′)
------------------------------------------------------------------------
{1}   a′ + length(bl′) ≤ floor(expt(2, size′))(a′) + expt(2, size′)
{2}   cons?(head(bl′, expt(2, size′) − rem(expt(2, size′))(offset(a′))))
```

Expanding the definition of head,
which is trivially true.
This completes the proof of `split_no_null.2.3`.
Q.E.D.

C.118.7 List_Split.split_null

Terse proof for `split_null`.
`split_null`:

```
------------------------------------------------------------------------
{1}   ∀ (size: nat, a: Address): split(size, a, null) = null
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_null`.
Q.E.D.

C.118.8 List_Split.split_type_TCC1

Terse proof for `split_type_TCC1`.

split_type_TCC1:

{1} Mem?(min_linear‘type_of)

Expanding the definition of min_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of split_type_TCC1.
Q.E.D.

C.118.9 List_Split.split_type_TCC2

Terse proof for split_type_TCC2.

split_type_TCC2:

{1} ∀ (a: Address): reg_base(min_linear)(type_of(a)) ≤ a ⊃ Mem?(max_linear‘type_of)

Expanding the definition of max_linear,
Expanding the definition of Mem,
which is trivially true.
This completes the proof of split_type_TCC2.
Q.E.D.

C.118.10 List_Split.split_type

Terse proof for split_type.

split_type:

{1} ∀ (size: nat, a: Address, bl: list[Byte], n: nat):
reg_base(min_linear)(type_of(a)) ≤ a ∧
a + length(bl) ≤ reg_size(max_linear)(type_of(a))
⊃
every(λ (t: [Address, list[Byte]]):
type_of(t‘1) = type_of(a) ∧
reg_base(min_linear)(type_of(a)) ≤ t‘1 ∧
t‘1 < reg_size(max_linear)(type_of(a)))
(split(min_page, a, bl))

Repeatedly Skolemizing and flattening,
Hiding formulas: -2,
Case splitting on bl!1 = null,
we get 2 subgoals:

`split_type.1:`

{-1} $\mathrm{bl}' = \mathrm{null}$
{-2} $\mathrm{size}' \geq 0$
{-3} $n' \geq 0$
{-4} $\mathrm{reg\_base}(\mathrm{min\_linear})(\mathrm{type\_of}(a')) \leq a'$
{-5} $a' + \mathrm{length}(\mathrm{bl}') \leq \mathrm{reg\_size}(\mathrm{max\_linear})(\mathrm{type\_of}(a'))$

{1} $\mathrm{every}(\lambda\ (t\colon [\mathrm{Address}, \mathrm{list}[\mathrm{Byte}]])\colon$
$\mathrm{type\_of}(t\text{`}1) = \mathrm{type\_of}(a') \wedge$
$\mathrm{reg\_base}(\mathrm{min\_linear})(\mathrm{type\_of}(a')) \leq t\text{`}1 \wedge$
$t\text{`}1 < \mathrm{reg\_size}(\mathrm{max\_linear})(\mathrm{type\_of}(a'))$
$(\mathrm{split}(\mathrm{min\_page}, a', \mathrm{bl}'))$

Using lemma split_null,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_type.1`.
`split_type.2:`

{-1} $\mathrm{size}' \geq 0$
{-2} $n' \geq 0$
{-3} $\mathrm{reg\_base}(\mathrm{min\_linear})(\mathrm{type\_of}(a')) \leq a'$
{-4} $a' + \mathrm{length}(\mathrm{bl}') \leq \mathrm{reg\_size}(\mathrm{max\_linear})(\mathrm{type\_of}(a'))$

{1} $\mathrm{bl}' = \mathrm{null}$
{2} $\mathrm{every}(\lambda\ (t\colon [\mathrm{Address}, \mathrm{list}[\mathrm{Byte}]])\colon$
$\mathrm{type\_of}(t\text{`}1) = \mathrm{type\_of}(a') \wedge$
$\mathrm{reg\_base}(\mathrm{min\_linear})(\mathrm{type\_of}(a')) \leq t\text{`}1 \wedge$
$t\text{`}1 < \mathrm{reg\_size}(\mathrm{max\_linear})(\mathrm{type\_of}(a'))$
$(\mathrm{split}(\mathrm{min\_page}, a', \mathrm{bl}'))$

Using lemma split_range,
Using lemma split_no_null,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma every_implied,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`split_type.2.1:`

{-1} $\mathrm{cons?}(\mathrm{bl}')$
{-2} $\mathrm{every}(\lambda\ (a_2\colon \mathrm{Address}, \mathrm{bl2}\colon \mathrm{list}[\mathrm{Byte}])\colon \mathrm{cons?}(\mathrm{bl2}))(\mathrm{split}(\mathrm{min\_page}, a', \mathrm{bl}'))$
{-3} $\mathrm{every}(\lambda\ (a_2\colon \mathrm{Address}, \mathrm{bl2}\colon \mathrm{list}[\mathrm{Byte}])\colon$
$a' \leq a_2 \wedge a_2 + \mathrm{length}(\mathrm{bl2}) \leq a' + \mathrm{length}(\mathrm{bl}'))$
$(\mathrm{split}(\mathrm{min\_page}, a', \mathrm{bl}'))$
{-4} $\mathrm{size}' \geq 0$
{-5} $n' \geq 0$
{-6} $\mathrm{reg\_base}(\mathrm{min\_linear})(\mathrm{type\_of}(a')) \leq a'$
{-7} $a' + \mathrm{length}(\mathrm{bl}') \leq \mathrm{reg\_size}(\mathrm{max\_linear})(\mathrm{type\_of}(a'))$

{1} $\mathrm{every}(\lambda\ (a_2\colon \mathrm{Address}, \mathrm{bl2}\colon \mathrm{list}[\mathrm{Byte}])\colon$
$\mathrm{cons?}(\mathrm{bl2}) \wedge a' \leq a_2 \wedge a_2 + \mathrm{length}(\mathrm{bl2}) \leq a' + \mathrm{length}(\mathrm{bl}'))$
$(\mathrm{split}(\mathrm{min\_page}, a', \mathrm{bl}'))$
{2} $\mathrm{every}(\lambda\ (t\colon [\mathrm{Address}, \mathrm{list}[\mathrm{Byte}]])\colon$
$\mathrm{type\_of}(t\text{`}1) = \mathrm{type\_of}(a') \wedge$
$\mathrm{reg\_base}(\mathrm{min\_linear})(\mathrm{type\_of}(a')) \leq t\text{`}1 \wedge$
$t\text{`}1 < \mathrm{reg\_size}(\mathrm{max\_linear})(\mathrm{type\_of}(a'))$
$(\mathrm{split}(\mathrm{min\_page}, a', \mathrm{bl}'))$

Using lemma every_conjunct_left,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `split_type.2.1`.
`split_type.2.2`:

```
{-1}  cons?(bl')
{-2}  every(λ (a2: Address, bl2: list[Byte]): cons?(bl2))(split(min_page, a', bl'))
{-3}  every(λ (a2: Address, bl2: list[Byte]):
                a' ≤ a2 ∧ a2 + length(bl2) ≤ a' + length(bl'))
            (split(min_page, a', bl'))
{-4}  size' ≥ 0
{-5}  n' ≥ 0
{-6}  reg_base(min_linear)(type_of(a')) ≤ a'
{-7}  a' + length(bl') ≤ reg_size(max_linear)(type_of(a'))
------------------------------------------------------------------------
{1}   ∀ (t_1: [Address, list[Byte]]):
        cons?(t_1`2) ∧ a' ≤ t_1`1 ∧ t_1`1 + length(t_1`2) ≤ a' + length(bl')
         ⊃
         type_of(t_1`1) = type_of(a') ∧
          reg_base(min_linear)(type_of(a')) ≤ t_1`1 ∧
           t_1`1 < reg_size(max_linear)(type_of(a'))
{2}   every(λ (t: [Address, list[Byte]]):
              type_of(t`1) = type_of(a') ∧
               reg_base(min_linear)(type_of(a')) ≤ t`1 ∧
                t`1 < reg_size(max_linear)(type_of(a')))
            (split(min_page, a', bl'))
```

Hiding formulas: (-2 -3 2),
Repeatedly Skolemizing and flattening,
Expanding the definition of <=,
Expanding the definition of <,
Expanding the definition of +,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of length,
Keeping (-5 -12 1) and hiding *,
Adding type constraints for length(cdr(t!1`2)),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `split_type.2.2`.
Q.E.D.

C.119 Proofs for LogicalExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.120 Proofs for LvalueToRvalueConversion (conversions.pvs)

This theory contains no provable formal statements.

C.121 Proofs for Memory (memory.pvs)

C.121.1 Memory.memory_write_list_nse_TCC1

Terse proof for `memory_write_list_nse_TCC1`.

`memory_write_list_nse_TCC1`:

{1} ∀ (bl: list[Byte], b: Byte, rl: list[Byte]): bl = cons(b, rl) ⊃ cons?[Byte](bl)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_write_list_nse_TCC1`.
Q.E.D.

C.121.2 Memory.memory_write_list_nse_TCC2

Terse proof for `memory_write_list_nse_TCC2`.

`memory_write_list_nse_TCC2`:

{1} ∀ (bl: list[Byte], b: Byte, rl: list[Byte]):
bl = cons(b, rl) ⊃ length[Byte](cdr[Byte](bl)) < length[Byte](bl)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_write_list_nse_TCC2`.
Q.E.D.

C.121.3 Memory.memory_read_list_nse_TCC1

Terse proof for `memory_read_list_nse_TCC1`.

`memory_read_list_nse_TCC1`:

{1} ∀ (size: nat): size > 0 ⊃ size − 1 ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_read_list_nse_TCC1`.
Q.E.D.

C.121.4 Memory.memory_read_list_nse_TCC2

Terse proof for `memory_read_list_nse_TCC2`.

`memory_read_list_nse_TCC2`:

{1} ∀ (size: nat): size > 0 ⊃ size − 1 < size

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_read_list_nse_TCC2`.
Q.E.D.

C.121.5 Memory.memory_read_list_ok_length

Terse proof for `memory_read_list_ok_length`.

memory_read_list_ok_length:

{1} ∀ (addr: Address, pm: Memory_struct, s: State, size: nat):
OK?(memory_read_list_nse(pm)(addr, size)(s)) ⊃
length(data(memory_read_list_nse(pm)(addr, size)(s))) = size

Inducting on size on formula 1,
we get 2 subgoals:
memory_read_list_ok_length.1:

{1} ∀ (addr: Address, pm: Memory_struct, s: State):
OK?(memory_read_list_nse(pm)(addr, 0)(s)) ⊃
length(data(memory_read_list_nse(pm)(addr, 0)(s))) = 0

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_read_list_nse,
Expanding the definition of ok_result,
Expanding the definition of length,
which is trivially true.
This completes the proof of memory_read_list_ok_length.1.
memory_read_list_ok_length.2:

{1} ∀ j:
(∀ (addr: Address, pm: Memory_struct, s: State):
OK?(memory_read_list_nse(pm)(addr, j)(s)) ⊃
length(data(memory_read_list_nse(pm)(addr, j)(s))) = j)
⊃
(∀ (addr: Address, pm: Memory_struct, s: State):
OK?(memory_read_list_nse(pm)(addr, $j+1$)(s)) ⊃
length(data(memory_read_list_nse(pm)(addr, $j+1$)(s))) = $j+1$)

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_read_list_nse,
Expanding the definition of ##,
Expanding the definition of ok_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of length,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of memory_read_list_ok_length.2.
Q.E.D.

C.121.6 Memory.memory_read_transformers_memory_read

Terse proof for memory_read_transformers_memory_read.
memory_read_transformers_memory_read:

{1} ∀ (addresses: PRED⌊Address⌋, pm: Memory_struct, addr: Address):
addresses(addr) ⊃
memory_read_transformers(pm, addresses)(expr_2_super(memory_read(pm)(addr)))

Repeatedly Skolemizing and flattening,

Expanding the definition of memory_read_transformers,
Instantiating quantified variables,
This completes the proof of `memory_read_transformers_memory_read`.
Q.E.D.

C.121.7 Memory.memory_read_transformers_mono

Terse proof for `memory_read_transformers_mono`.
`memory_read_transformers_mono:`

{1} ∀ (pm: Memory_struct, addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ⊃
(memory_read_transformers(pm, addresses_1) ⊆ memory_read_transformers(pm, addresses_2))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_read_transformers_mono`.
Q.E.D.

C.121.8 Memory.memory_read_transformers_union

Terse proof for `memory_read_transformers_union`.
`memory_read_transformers_union:`

{1} ∀ (pm: Memory_struct, addresses_1, addresses_2: PRED[Address]):
(memory_read_transformers(pm, addresses_1) ∪ memory_read_transformers(pm, addresses_2))
= memory_read_transformers(pm, (addresses_1 ∪ addresses_2))

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_read_transformers,
Expanding the definition of union,
Expanding the definition of member,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_read_transformers_union`.
Q.E.D.

C.121.9 Memory.memory_write_transformers_memory_write

Terse proof for `memory_write_transformers_memory_write`.
`memory_write_transformers_memory_write:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct, addr: Address, *b*: Byte):
addresses(addr) ⊃
memory_write_transformers(pm, addresses)(expr_2_super(memory_write(pm)(addr, *b*)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_write_transformers_memory_write`.
Q.E.D.

C.121.10 Memory.memory_write_transformers_mono

Terse proof for `memory_write_transformers_mono`.
`memory_write_transformers_mono`:

{1} ∀ (pm: Memory_struct, addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ⊃
(memory_write_transformers(pm, addresses_1) ⊆ memory_write_transformers(pm, addresses_2))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `memory_write_transformers_mono`.
Q.E.D.

C.121.11 Memory.memory_read_side_effect_super_transformers_memory_read_side_effect

Terse proof for `memory_read_side_effect_super_transformers_memory_read_side_effect`.
`memory_read_side_effect_super_transformers_memory_read_side_effect`:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct, a: Address, bl: list[Byte], cp: bool):
(address_block(a, length(bl)) ⊆ addresses) ⊃
memory_read_side_effect_super_transformers(pm, addresses)
(expr_2_super(memory_read_side_effect(pm)
(a, bl, cp)))

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_read_side_effect_super_transformers,
Instantiating quantified variables,
This completes the proof of `memory_read_side_effect_super_transformers_memory_read_side_effect`.
Q.E.D.

C.121.12 Memory.memory_read_side_effect_super_transformers_mono

Terse proof for `memory_read_side_effect_super_transformers_mono`.
`memory_read_side_effect_super_transformers_mono`:

{1} ∀ (pm: Memory_struct, addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ⊃
(memory_read_side_effect_super_transformers(pm, addresses_1) ⊆ memory_read_side_effect_super_transform

Repeatedly Skolemizing and flattening,
Expanding the definition of subset?,
Expanding the definition of member,
Expanding the definition of memory_read_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on subset_transitive,
This completes the proof of `memory_read_side_effect_super_transformers_mono`.

Q.E.D.

C.121.13 Memory.memory_write_side_effect_super_transformers_memory_write_side_effect

Terse proof for `memory_write_side_effect_super_transformers_memory_write_side_effect`.

`memory_write_side_effect_super_transformers_memory_write_side_effect:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct, a: Address, bl: list[Byte], cp: bool):
(address_block(a, length(bl)) ⊆ addresses) ⊃
memory_write_side_effect_super_transformers(pm, addresses)
(expr_2_super(memory_write_side_effect(pm)
(a, bl, cp)))

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_side_effect_super_transformers,
Instantiating quantified variables,
This completes the proof of `memory_write_side_effect_super_transformers_memory_write_side_effect`.
Q.E.D.

C.121.14 Memory.memory_write_side_effect_super_transformers_mono

Terse proof for `memory_write_side_effect_super_transformers_mono`.

`memory_write_side_effect_super_transformers_mono:`

{1} ∀ (pm: Memory_struct, addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ⊃
(memory_write_side_effect_super_transformers(pm, addresses_1) ⊆ memory_write_side_effect_sup

Repeatedly Skolemizing and flattening,
Expanding the definition of subset?,
Expanding the definition of member,
Expanding the definition of memory_write_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on subset_transitive,
This completes the proof of `memory_write_side_effect_super_transformers_mono`.
Q.E.D.

C.121.15 Memory.side_effect_content_unchanged_content

Terse proof for `side_effect_content_unchanged_content`.

side_effect_content_unchanged_content:

{1} ∀ (se_transformer:
[[Address, list[Byte], bool] → [State → ExprResult[State, list[Byte]]]],
states: PRED[State], addresses: PRED[Address], s: State, a: Address, bl: list[Byte],
cp: bool):
side_effect_content_unchanged(addresses, states, se_transformer) ∧
states(s) ∧
(address_block(a, length(bl)) ⊆ addresses) ∧
OK?(se_transformer(a, bl, cp)(s))
⊃ data(se_transformer(a, bl, cp)(s)) = bl

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of side_effect_content_unchanged_content.

Q.E.D.

C.121.16 Memory.side_effect_content_unchanged_mono

Terse proof for side_effect_content_unchanged_mono.

side_effect_content_unchanged_mono:

{1} ∀ (se_transformer:
[[Address, list[Byte], bool] → [State → ExprResult[State, list[Byte]]]],
states: PRED[State], addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ∧
side_effect_content_unchanged(addresses_2, states, se_transformer)
⊃ side_effect_content_unchanged(addresses_1, states, se_transformer)

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2 -5 1) and hiding *,

Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets addresses!1, c gets addresses!2,

This completes the proof of side_effect_content_unchanged_mono.

Q.E.D.

C.121.17 Memory.side_effect_content_unchanged_composition

Terse proof for side_effect_content_unchanged_composition.

`side_effect_content_unchanged_composition:`

```
{1}  ∀ (addresses: PRED[Address],
          se_transformer1,
          se_transformer2:
            [[Address, list[Byte], bool] → [State → ExprResult[State, list[Byte]]]],
          states: PRED[State]):
        transformer_invariant?(states, se_super_transformers(addresses, se_transformer1)) ∧
         side_effect_content_unchanged(addresses, states, se_transformer1) ∧
          side_effect_content_unchanged(addresses, states, se_transformer2)
         ⊃
         side_effect_content_unchanged(addresses, states,
                                       λ (a: Address, bl: list[Byte], cp: bool):
                                         se_transformer1(a, bl, cp) ##
                                          (λ (bl1: list[Byte]): se_transformer2(a, bl1, cp)))
```

Expanding the definition of side_effect_content_unchanged,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in -3 with the terms: (s!1 a!1 bl!1 cp!1),

Expanding the definition of ##,

Installing automatic rewrites from: has_next_state

Using lemma expr_transformer_invariant_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`side_effect_content_unchanged_composition.1:`

```
{-1}  transformer_invariant?(states′, se_super_transformers(addresses′, se_transformer1′))
{-2}  states′(s′)
{-3}  states′(state(se_transformer1′(a′, bl′, cp′)(s′)))
{-4}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-5}  (address_block(a′, length(bl′)) ⊆ addresses′)
{-6}  data(se_transformer1′(a′, bl′, cp′)(s′)) = bl′
{-7}  ∀ (s: State, a: Address, bl: list[Byte], cp: bool):
         states′(s) ∧
          (address_block(a, length(bl)) ⊆ addresses′) ∧
           OK?(se_transformer2′(a, bl, cp)(s))
          ⊃ data(se_transformer2′(a, bl, cp)(s)) = bl
{-8}  OK?(se_transformer1′(a′, bl′, cp′)(s′))
{-9}  OK?(se_transformer2′(a′, data(se_transformer1′(a′, bl′, cp′)(s′)), cp′)
                           (state(se_transformer1′(a′, bl′, cp′)(s′))))
------------------------------------------------------------------------
{1}  data(se_transformer2′(a′, data(se_transformer1′(a′, bl′, cp′)(s′)), cp′)
                            (state(se_transformer1′(a′, bl′, cp′)(s′))))
      = bl′
```

Instantiating the top quantifier in -7 with the terms: (state(se_transformer1!1(a!1, bl!1, cp!1)(s!1)) a!1 bl!1 cp!1),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `side_effect_content_unchanged_composition.1`.

side_effect_content_unchanged_composition.2:

{-1} transformer_invariant?(states′, se_super_transformers(addresses′, se_transformer1′))
{-2} states′(s′)
{-3} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-4} (address_block(a′, length(bl′)) ⊆ addresses′)
{-5} data(se_transformer1′(a′, bl′, cp′)(s′)) = bl′
{-6} ∀ (s: State, a: Address, bl: list[Byte], cp: bool):
states′(s) ∧
(address_block(a, length(bl)) ⊆ addresses′) ∧
OK?(se_transformer2′(a, bl, cp)(s))
⊃ data(se_transformer2′(a, bl, cp)(s)) = bl
{-7} OK?(se_transformer1′(a′, bl′, cp′)(s′))
{-8} OK?(se_transformer2′(a′, data(se_transformer1′(a′, bl′, cp′)(s′)), cp′)
(state(se_transformer1′(a′, bl′, cp′)(s′))))

{1} se_super_transformers(addresses′, se_transformer1′)
(expr_2_super(se_transformer1′(a′, bl′, cp′)))
{2} data(se_transformer2′(a′, data(se_transformer1′(a′, bl′, cp′)(s′)), cp′)
(state(se_transformer1′(a′, bl′, cp′)(s′))))
= bl′

Expanding the definition of se_super_transformers,
Instantiating quantified variables,
This completes the proof of side_effect_content_unchanged_composition.2.
Q.E.D.

C.122 Proofs for Memory_Change (memory.pvs)

C.122.1 Memory_Change.unchanged_memory_invariant?_TCC1

Terse proof for unchanged_memory_invariant?_TCC1.

unchanged_memory_invariant?_TCC1:

{1} ∀ (pm: Memory_struct[State], states: PRED[State],
transformers: PRED[[State → SuperResult[State]]], addresses: PRED[Address]):
transformer_invariant?(states, transformers) ⊃
(∀ (s: State, q: [State → SuperResult[State]], a: Address):
OK?(memory_read(pm)(a)(s)) ∧
OK?(q(s)) ∧ addresses(a) ∧ transformers(q) ∧ states(s)
⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of unchanged_memory_invariant?_TCC1.
Q.E.D.

C.122.2 Memory_Change.unchanged_memory_invariant_unchanged_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_TCC1`.

`unchanged_memory_invariant_unchanged_TCC1:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State, q: [State → SuperResult[State]], a: Address): OK?(memory_read(pm)(a)(s)) ∧ OK?(q(s)) ∧ addresses(a) ∧ transformers(q) ∧ states(s) ∧ unchanged_memory_invariant?(pm, states, transformers, addresses) ⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_TCC1`.

Q.E.D.

C.122.3 Memory_Change.unchanged_memory_invariant_unchanged

Terse proof for `unchanged_memory_invariant_unchanged`.

`unchanged_memory_invariant_unchanged:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State, q: [State → SuperResult[State]], a: Address): unchanged_memory_invariant?(pm, states, transformers, addresses) ∧ states(s) ∧ transformers(q) ∧ addresses(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm)(a)(s)) ∧ OK?(memory_read(pm)(a)(state(q(s)))) ⊃ data(memory_read(pm)(a)(state(q(s)))) = data(memory_read(pm)(a)(s))

Repeatedly Skolemizing and flattening,

Expanding the definition of unchanged_memory_invariant?,

Applying disjunctive simplification to flatten sequent,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged`.

Q.E.D.

C.122.4 Memory_Change.unchanged_memory_invariant_invariant

Terse proof for `unchanged_memory_invariant_invariant`.

unchanged_memory_invariant_invariant:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
transformers: PRED[[State → SuperResult[State]]]):
unchanged_memory_invariant?(pm, states, transformers, addresses) ⊃
transformer_invariant?(states, transformers)

Repeatedly Skolemizing and flattening,
Expanding the definition of unchanged_memory_invariant?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of unchanged_memory_invariant_invariant.
Q.E.D.

C.122.5 Memory_Change.unchanged_memory_invariant_mono

Terse proof for unchanged_memory_invariant_mono.
unchanged_memory_invariant_mono:

{1} ∀ (pm: Memory_struct[State], states: PRED[State],
transformers_1, transformers_2: PRED[[State → SuperResult[State]]],
addresses_1, addresses_2: PRED[Address]):
(transformers_1 ⊆ transformers_2) ∧
(addresses_1 ⊆ addresses_2) ∧
unchanged_memory_invariant?(pm, states, transformers_2, addresses_2)
⊃ unchanged_memory_invariant?(pm, states, transformers_1, addresses_1)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of unchanged_memory_invariant_mono.
Q.E.D.

C.122.6 Memory_Change.unchanged_memory_invariant_next_ok_TCC1

Terse proof for unchanged_memory_invariant_next_ok_TCC1.
unchanged_memory_invariant_next_ok_TCC1:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states, transformers, addresses) ∧
states(s) ∧ transformers(q) ∧ has_next_state(q(s))
⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of unchanged_memory_invariant_next_ok_TCC1.
Q.E.D.

C.122.7 Memory_Change.unchanged_memory_invariant_next_ok

Terse proof for unchanged_memory_invariant_next_ok.

`unchanged_memory_invariant_next_ok:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State, q: [State → SuperResult[State]]): unchanged_memory_invariant?(pm, states, transformers, addresses) ∧ states(s) ∧ transformers(q) ∧ has_next_state(q(s)) ⊃ states(state(q(s)))

Repeatedly Skolemizing and flattening,

Expanding the definition of unchanged_memory_invariant?,

Applying disjunctive simplification to flatten sequent,

Using lemma super_transformer_invariant_next_ok[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_next_ok`.

Q.E.D.

C.122.8 Memory_Change.unchanged_memory_invariant_union_transformers

Terse proof for `unchanged_memory_invariant_union_transformers`.

`unchanged_memory_invariant_union_transformers:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], transformers_1, transformers_2: PRED[[State → SuperResult[State]]]): unchanged_memory_invariant?(pm, states, transformers_1, addresses) ∧ unchanged_memory_invariant?(pm, states, transformers_2, addresses) ⊃ unchanged_memory_invariant?(pm, states, (transformers_1 ∪ transformers_2), addresses)

Installing automatic rewrites from: member

Repeatedly Skolemizing and flattening,

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

`unchanged_memory_invariant_union_transformers.1:`

{-1} transformer_invariant?(states′, transformers′)
{-2} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers′(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
{-3} transformer_invariant?(states′, transformers″)
{-4} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers″(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

{1} transformer_invariant?(states′, (transformers′ ∪ transformers″))

Rewriting using transformer_invariant_union_transformers, matching in *,

This completes the proof of `unchanged_memory_invariant_union_transformers.1`.

`unchanged_memory_invariant_union_transformers.2:`

{-1} transformer_invariant?(states′, transformers′)
{-2} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers′(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
{-3} transformer_invariant?(states′, transformers″)
{-4} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers″(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

{1} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
union(transformers′, transformers″)(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

Hiding formulas: (-1 -3),

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Splitting conjunctions,

we get 2 subgoals:

`unchanged_memory_invariant_union_transformers.2.1:`

{-1} $(q' \in \text{transformers}')$
{-2} $\text{states}'(s')$
{-3} $\text{addresses}'(a')$
{-4} $\text{OK?}(q'(s'))$
{-5} $\text{OK?}(\text{memory\_read}(\text{pm}')(a')(s'))$
{-6} $\text{OK?}(\text{memory\_read}(\text{pm}')(a')(\text{state}(q'(s'))))$
{-7} $\forall\ (s\text{: State},\ q\text{: } [\text{State} \to \text{SuperResult}[\text{State}]],\ a\text{: Address}):$
$\text{states}'(s) \wedge$
$\text{transformers}'(q) \wedge$
$\text{addresses}'(a) \wedge$
$\text{OK?}(q(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) = \text{data}(\text{memory\_read}(\text{pm}')(a)(s))$
{-8} $\forall\ (s\text{: State},\ q\text{: } [\text{State} \to \text{SuperResult}[\text{State}]],\ a\text{: Address}):$
$\text{states}'(s) \wedge$
$\text{transformers}''(q) \wedge$
$\text{addresses}'(a) \wedge$
$\text{OK?}(q(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) = \text{data}(\text{memory\_read}(\text{pm}')(a)(s))$

{1} $\text{data}(\text{memory\_read}(\text{pm}')(a')(\text{state}(q'(s')))) =$
$\text{data}(\text{memory\_read}(\text{pm}')(a')(s'))$

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_union_transformers.2.1`.

`unchanged_memory_invariant_union_transformers.2.2:`

{-1} $(q' \in \text{transformers}'')$
{-2} $\text{states}'(s')$
{-3} $\text{addresses}'(a')$
{-4} $\text{OK?}(q'(s'))$
{-5} $\text{OK?}(\text{memory\_read}(\text{pm}')(a')(s'))$
{-6} $\text{OK?}(\text{memory\_read}(\text{pm}')(a')(\text{state}(q'(s'))))$
{-7} $\forall\ (s\text{: State},\ q\text{: }[\text{State} \to \text{SuperResult}[\text{State}]],\ a\text{: Address}):$
$\text{states}'(s)\ \wedge$
$\text{transformers}'(q)\ \wedge$
$\text{addresses}'(a)\ \wedge$
$\text{OK?}(q(s))\ \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s))\ \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) = \text{data}(\text{memory\_read}(\text{pm}')(a)(s))$
{-8} $\forall\ (s\text{: State},\ q\text{: }[\text{State} \to \text{SuperResult}[\text{State}]],\ a\text{: Address}):$
$\text{states}'(s)\ \wedge$
$\text{transformers}''(q)\ \wedge$
$\text{addresses}'(a)\ \wedge$
$\text{OK?}(q(s))\ \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s))\ \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) = \text{data}(\text{memory\_read}(\text{pm}')(a)(s))$

{1} $\text{data}(\text{memory\_read}(\text{pm}')(a')(\text{state}(q'(s')))) =$
$\text{data}(\text{memory\_read}(\text{pm}')(a')(s'))$

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unchanged_memory_invariant_union_transformers.2.2`.
Q.E.D.

C.122.9 Memory_Change.unchanged_memory_invariant_union_addresses

Terse proof for `unchanged_memory_invariant_union_addresses`.

`unchanged_memory_invariant_union_addresses:`

{1} $\forall\ (\text{pm: Memory\_struct}[\text{State}],\ \text{states: PRED}[\text{State}],$
$\text{transformers: PRED}[[\text{State} \to \text{SuperResult}[\text{State}]]],$
$\text{addresses\_1, addresses\_2: PRED}[\text{Address}]):$
$\text{unchanged\_memory\_invariant?}(\text{pm, states, transformers, addresses\_1})\ \wedge$
$\text{unchanged\_memory\_invariant?}(\text{pm, states, transformers, addresses\_2})$
$\supset \text{unchanged\_memory\_invariant?}(\text{pm, states, transformers, (addresses\_1} \cup \text{addresses\_2))}$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_invariant_union_addresses`.
Q.E.D.

C.122.10 Memory_Change.unchanged_memory_invariant_empty

Terse proof for `unchanged_memory_invariant_empty`.

`unchanged_memory_invariant_empty:`

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
          transformers: PRED[[State → SuperResult[State]]]):
       empty?(states) ∨ empty?(transformers) ⊃
        unchanged_memory_invariant?(pm, states, transformers, addresses)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_invariant_empty`.
Q.E.D.

C.122.11 Memory_Change.unchanged_memory_invariant_all_transformers

Terse proof for `unchanged_memory_invariant_all_transformers`.

`unchanged_memory_invariant_all_transformers:`

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
          transformers: PRED[[State → SuperResult[State]]]):
       (∀ (q: [State → SuperResult[State]]):
          transformers(q) ⊃ unchanged_memory_invariant?(pm, states, singleton(q), ad-
    dresses))
        ⊃ unchanged_memory_invariant?(pm, states, transformers, addresses)
```

Expanding the definition of unchanged_memory_invariant?,
Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`unchanged_memory_invariant_all_transformers.1:`

```
{-1}  ∀ (q: [State → SuperResult[State]]):
        transformers′(q) ⊃
         transformer_invariant?(states′, singleton(q)) ∧
          (∀ (s: State, q_1: [State → SuperResult[State]], a: Address):
             states′(s) ∧
              singleton(q)(q_1) ∧
               addresses′(a) ∧
                OK?(q_1(s)) ∧
                 OK?(memory_read(pm′)(a)(s)) ∧
                  OK?(memory_read(pm′)(a)(state(q_1(s))))
             ⊃
             data(memory_read(pm′)(a)(state(q_1(s)))) =
              data(memory_read(pm′)(a)(s)))
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′, transformers′)
```

Rewriting using transformer_invariant_all_transformers, matching in *,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_all_transformers.1`.

unchanged_memory_invariant_all_transformers.2:

{-1} ∀ (q: [State → SuperResult[State]]):
transformers′(q) ⊃
transformer_invariant?(states′, singleton(q)) ∧
(∀ (s: State, q_1: [State → SuperResult[State]], a: Address):
states′(s) ∧
singleton(q)(q_1) ∧
addresses′(a) ∧
OK?(q_1(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q_1(s))))
⊃
data(memory_read(pm′)(a)(state(q_1(s)))) =
data(memory_read(pm′)(a)(s)))

{1} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers′(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of singleton,
which is trivially true.
This completes the proof of unchanged_memory_invariant_all_transformers.2.
Q.E.D.

C.123 Proofs for Memory_Change_2 (memory.pvs)

C.123.1 Memory_Change_2.unchanged_memory_write_invariant_mono_addresses

Terse proof for unchanged_memory_write_invariant_mono_addresses.
unchanged_memory_write_invariant_mono_addresses:

{1} ∀ (pm: Memory_struct[State], states: PRED[State],
addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ∧ unchanged_memory_write_invariant?(pm, states, addresses_2)
⊃ unchanged_memory_write_invariant?(pm, states, addresses_1)

Repeatedly Skolemizing and flattening,
Expanding the definition of unchanged_memory_write_invariant?,
Repeatedly Skolemizing and flattening,

Instantiating quantified variables,
Applying propositional simplification,
we get 2 subgoals:
`unchanged_memory_write_invariant_mono_addresses.1:`

{-1} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_write(pm′)(waddr′, b′))), (addresses″ \ {waddr′}))
{-2} $b' < \text{max\_byte}$
{-3} $(\text{addresses}' \subseteq \text{addresses}'')$
{-4} addresses′(waddr′)

{1} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_write(pm′)(waddr′, b′))), (addresses′ \ {waddr′}))

Using lemma unchanged_memory_invariant_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_equal, matching in *,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_write_invariant_mono_addresses.1`.
`unchanged_memory_write_invariant_mono_addresses.2:`

{-1} $b' < \text{max\_byte}$
{-2} $(\text{addresses}' \subseteq \text{addresses}'')$
{-3} addresses′(waddr′)

{1} addresses″(waddr′)
{2} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_write(pm′)(waddr′, b′))), (addresses′ \ {waddr′}))

Installing automatic rewrites from: subset? singleton member
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of `unchanged_memory_write_invariant_mono_addresses.2`.
Q.E.D.

C.123.2 Memory_Change_2.unchanged_memory_write_invariant_transformer_invariant

Terse proof for `unchanged_memory_write_invariant_transformer_invariant`.
`unchanged_memory_write_invariant_transformer_invariant:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State]):
unchanged_memory_write_invariant?(pm, states, addresses) ⊃
transformer_invariant?(states, memory_write_transformers(pm, addresses))

Repeatedly Skolemizing and flattening,
Expanding the definition of unchanged_memory_write_invariant?,
Expanding the definition of unchanged_memory_invariant?,
Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -5,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Applying disjunctive simplification to flatten sequent,

Expanding the definition of singleton,

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_write_invariant_transformer_invariant`.

Q.E.D.

C.123.3 Memory_Change_2.unchanged_memory_write_invariant_unchanged_memory_invari

Terse proof for `unchanged_memory_write_invariant_unchanged_memory_invariant`.

`unchanged_memory_write_invariant_unchanged_memory_invariant:`

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State],
          all_addresses, unchanged_addresses, changed_addresses: PRED[Address]):
       unchanged_memory_write_invariant?(pm, states, all_addresses) ∧
        (unchanged_addresses ⊆ all_addresses) ∧
         (changed_addresses ⊆ all_addresses) ∧ disjoint?(unchanged_addresses, changed_addresses)
        ⊃
        unchanged_memory_invariant?(pm, states, memory_write_transformers(pm, changed_addresse
                                                unchanged_addresses)
```

Repeatedly Skolemizing and flattening,

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

`unchanged_memory_write_invariant_unchanged_memory_invariant.1:`

```
{-1}  unchanged_memory_write_invariant?(pm′, states′, all_addresses′)
{-2}  (unchanged_addresses′ ⊆ all_addresses′)
{-3}  (changed_addresses′ ⊆ all_addresses′)
{-4}  disjoint?(unchanged_addresses′, changed_addresses′)
------------------------------------------------------------------------
 {1}  transformer_invariant?(states′, memory_write_transformers(pm′, changed_addresses′))
```

Rewriting using unchanged_memory_write_invariant_transformer_invariant, matching in *,

Hiding formulas: -2,

Forward chaining on unchanged_memory_write_invariant_mono_addresses,

This completes the proof of `unchanged_memory_write_invariant_unchanged_memory_invariant.1`.

`unchanged_memory_write_invariant_unchanged_memory_invariant.2:`

```
{-1}  unchanged_memory_write_invariant?(pm′, states′, all_addresses′)
{-2}  (unchanged_addresses′ ⊆ all_addresses′)
{-3}  (changed_addresses′ ⊆ all_addresses′)
{-4}  disjoint?(unchanged_addresses′, changed_addresses′)
------------------------------------------------------------------
{1}   ∀ (s: State, q: [State → SuperResult[State]], a: Address):
        states′(s) ∧
         memory_write_transformers(pm′, changed_addresses′)(q) ∧
          unchanged_addresses′(a) ∧
           OK?(q(s)) ∧
            OK?(memory_read(pm′)(a)(s)) ∧
             OK?(memory_read(pm′)(a)(state(q(s))))
         ⊃
         data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
```

Expanding the definition of memory_write_transformers,
Repeatedly Skolemizing and flattening,
Replacing using formula -4,
Expanding the definition of unchanged_memory_write_invariant?,
Expanding the definition of unchanged_memory_invariant?,
Instantiating quantified variables,
Applying propositional simplification,
we get 2 subgoals:

`unchanged_memory_write_invariant_unchanged_memory_invariant.2.1:`

```
{-1}  transformer_invariant?(states′, singleton(expr_2_super(memory_write(pm′)(a″, b′))))
{-2}  ∀ (s: State, q: [State → SuperResult[State]], a: Address):
        states′(s) ∧
         singleton(expr_2_super(memory_write(pm′)(a″, b′)))(q) ∧
          remove(a″, all_addresses′)(a) ∧
           OK?(q(s)) ∧
            OK?(memory_read(pm′)(a)(s)) ∧
             OK?(memory_read(pm′)(a)(state(q(s))))
         ⊃
         data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
{-3}  b′ < max_byte
{-4}  states′(s′)
{-5}  changed_addresses′(a″)
{-6}  q′ = expr_2_super(memory_write(pm′)(a″, b′))
{-7}  unchanged_addresses′(a′)
{-8}  OK?(expr_2_super(memory_write(pm′)(a″, b′))(s′))
{-9}  OK?(memory_read(pm′)(a′)(s′))
{-10} OK?(memory_read(pm′)
                     (a′)(state(expr_2_super(memory_write(pm′)(a″, b′))(s′))))
{-11} (unchanged_addresses′ ⊆ all_addresses′)
{-12} (changed_addresses′ ⊆ all_addresses′)
{-13} disjoint?(unchanged_addresses′, changed_addresses′)
------------------------------------------------------------------
{1}   data(memory_read(pm′)
                       (a′)(state(expr_2_super(memory_write(pm′)(a″, b′))(s′))))
        = data(memory_read(pm′)(a′)(s′))
```

Rewriting using -2, matching in *,
we get 2 subgoals:

`unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.1:`

{-1}	transformer_invariant?(states′, singleton(expr_2_super(memory_write(pm′)(a″, b′))))
{-2}	∀ (s: State, q: [State → SuperResult[State]], a: Address): states′(s) ∧ singleton(expr_2_super(memory_write(pm′)(a″, b′)))(q) ∧ remove(a″, all_addresses′)(a) ∧ OK?(q(s)) ∧ OK?(memory_read(pm′)(a)(s)) ∧ OK?(memory_read(pm′)(a)(state(q(s)))) ⊃ data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
{-3}	b′ < max_byte
{-4}	states′(s′)
{-5}	changed_addresses′(a″)
{-6}	q′ = expr_2_super(memory_write(pm′)(a″, b′))
{-7}	unchanged_addresses′(a′)
{-8}	OK?(expr_2_super(memory_write(pm′)(a″, b′))(s′))
{-9}	OK?(memory_read(pm′)(a′)(s′))
{-10}	OK?(memory_read(pm′)(a′)(state(expr_2_super(memory_write(pm′)(a″, b′))(s′))))
{-11}	(unchanged_addresses′ ⊆ all_addresses′)
{-12}	(changed_addresses′ ⊆ all_addresses′)
{-13}	disjoint?(unchanged_addresses′, changed_addresses′)
{1}	singleton(expr_2_super(memory_write(pm′)(a″, b′)))(expr_2_super(memory_write(pm′)(a″, b′)))
{2}	data(memory_read(pm′)(a′)(state(expr_2_super(memory_write(pm′)(a″, b′))(s′)))) = data(memory_read(pm′)(a′)(s′))

Expanding the definition of singleton,

which is trivially true.

This completes the proof of `unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.1`.

`unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.2:`

```
{-1}  transformer_invariant?(states', singleton(expr_2_super(memory_write(pm')(a'', b'))))
{-2}  ∀ (s: State, q: [State → SuperResult[State]], a: Address):
        states'(s) ∧
          singleton(expr_2_super(memory_write(pm')(a'', b')))(q) ∧
            remove(a'', all_addresses')(a) ∧
              OK?(q(s)) ∧
                OK?(memory_read(pm')(a)(s)) ∧
                  OK?(memory_read(pm')(a)(state(q(s))))
        ⊃
        data(memory_read(pm')(a)(state(q(s)))) = data(memory_read(pm')(a)(s))
{-3}  b' < max_byte
{-4}  states'(s')
{-5}  changed_addresses'(a'')
{-6}  q' = expr_2_super(memory_write(pm')(a'', b'))
{-7}  unchanged_addresses'(a')
{-8}  OK?(expr_2_super(memory_write(pm')(a'', b'))(s'))
{-9}  OK?(memory_read(pm')(a')(s'))
{-10} OK?(memory_read(pm')
                (a')(state(expr_2_super(memory_write(pm')(a'', b'))(s'))))
{-11} (unchanged_addresses' ⊆ all_addresses')
{-12} (changed_addresses' ⊆ all_addresses')
{-13} disjoint?(unchanged_addresses', changed_addresses')
-------------------------------------------------------------------------------
 {1}  remove(a'', all_addresses')(a')
 {2}  data(memory_read(pm')
                 (a')(state(expr_2_super(memory_write(pm')(a'', b'))(s'))))
        = data(memory_read(pm')(a')(s'))
```

Keeping (-5 -7 -11 -12 -13 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_write_invariant_unchanged_memory_invariant.2.1.2`.
`unchanged_memory_write_invariant_unchanged_memory_invariant.2.2:`

```
{-1}  b' < max_byte
{-2}  states'(s')
{-3}  changed_addresses'(a'')
{-4}  q' = expr_2_super(memory_write(pm')(a'', b'))
{-5}  unchanged_addresses'(a')
{-6}  OK?(expr_2_super(memory_write(pm')(a'', b'))(s'))
{-7}  OK?(memory_read(pm')(a')(s'))
{-8}  OK?(memory_read(pm')
                (a')(state(expr_2_super(memory_write(pm')(a'', b'))(s'))))
{-9}  (unchanged_addresses' ⊆ all_addresses')
{-10} (changed_addresses' ⊆ all_addresses')
{-11} disjoint?(unchanged_addresses', changed_addresses')
-------------------------------------------------------------------------------
 {1}  all_addresses'(a'')
 {2}  data(memory_read(pm')
                 (a')(state(expr_2_super(memory_write(pm')(a'', b'))(s'))))
        = data(memory_read(pm')(a')(s'))
```

Keeping (-3 -10 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_write_invariant_unchanged_memory_invariant.2.2`.

Q.E.D.

C.123.4 Memory_Change_2.unchanged_memory_invariant_union

Terse proof for `unchanged_memory_invariant_union`.

`unchanged_memory_invariant_union:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State],
ro_addr, rw_addr, mod_addr: PRED[Address]):
unchanged_memory_invariant?(pm, states, memory_write_transformers(pm, rw_addr), ro_addr
unchanged_memory_write_invariant?(pm, states, rw_addr) ∧ (mod_addr ⊆ rw_addr)
⊃
unchanged_memory_invariant?(pm, states, memory_write_transformers(pm, mod_addr),
((ro_addr ∪ rw_addr) \ mod_addr))

Repeatedly Skolemizing and flattening,

Case splitting on difference(union(ro_addr!1, rw_addr!1), mod_addr!1) = union(difference(ro_addr!1, mod_addr!1), difference(rw_addr!1, mod_addr!1)),

we get 2 subgoals:

`unchanged_memory_invariant_union.1:`

{-1} ((ro_addr′ ∪ rw_addr′) \ mod_addr′) =
((ro_addr′ \ mod_addr′) ∪ (rw_addr′ \ mod_addr′))
{-2} unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, rw_addr′),
ro_addr′)
{-3} unchanged_memory_write_invariant?(pm′, states′, rw_addr′)
{-4} (mod_addr′ ⊆ rw_addr′)

{1} unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, mod_addr′),
((ro_addr′ ∪ rw_addr′) \ mod_addr′))

Replacing using formula -1,

Hiding formulas: -1,

Rewriting using unchanged_memory_invariant_union_addresses, matching in *,

we get 2 subgoals:

`unchanged_memory_invariant_union.1.1:`

{-1} unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, rw_addr′),
ro_addr′)
{-2} unchanged_memory_write_invariant?(pm′, states′, rw_addr′)
{-3} (mod_addr′ ⊆ rw_addr′)

{1} unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, mod_addr′),
(ro_addr′ \ mod_addr′))
{2} unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, mod_addr′),
((ro_addr′ \ mod_addr′) ∪ (rw_addr′ \ mod_addr′)))

Hiding formulas: -2, 2,

Using lemma unchanged_memory_invariant_mono,

Installing automatic rewrites from: memory_write_transformers_mono

Simplifying, rewriting, and recording with decision procedures,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_invariant_union.1.1`.

`unchanged_memory_invariant_union.1.2:`

{-1}	unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, rw_addr′), ro_addr′)
{-2}	unchanged_memory_write_invariant?(pm′, states′, rw_addr′)
{-3}	(mod_addr′ ⊆ rw_addr′)
{1}	unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, mod_addr′), (rw_addr′ \ mod_addr′))
{2}	unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, mod_addr′), ((ro_addr′ \ mod_addr′) ∪ (rw_addr′ \ mod_addr′)))

Hiding formulas: -1, 2,

Using lemma unchanged_memory_write_invariant_unchanged_memory_invariant,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: difference_subset difference_disjoint_3

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_union.1.2`.

`unchanged_memory_invariant_union.2:`

{-1}	unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, rw_addr′), ro_addr′)
{-2}	unchanged_memory_write_invariant?(pm′, states′, rw_addr′)
{-3}	(mod_addr′ ⊆ rw_addr′)
{1}	((ro_addr′ ∪ rw_addr′) \ mod_addr′) = ((ro_addr′ \ mod_addr′) ∪ (rw_addr′ \ mod_addr′))
{2}	unchanged_memory_invariant?(pm′, states′, memory_write_transformers(pm′, mod_addr′), ((ro_addr′ ∪ rw_addr′) \ mod_addr′))

Keeping (-3 1) and hiding *,

Trying repeated skolemization, instantiation, if-lifting, and extensionality,

This completes the proof of `unchanged_memory_invariant_union.2`.

Q.E.D.

C.123.5 Memory_Change_2.transformer_invariant_write_list_nse

Terse proof for `transformer_invariant_write_list_nse`.

`transformer_invariant_write_list_nse:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
s: State):
transformer_invariant?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧ states(s)
⊃ result_pred(states)(expr_2_super(memory_write_list_nse(pm)(waddr, bl))(s))

For the top quantifier in 1, we introduce Skolem constants: (pm′ _ _ _ _),

Inducting on bl on formula 1,

we get 2 subgoals:

`transformer_invariant_write_list_nse.1:`

```
{1}  ∀ (states: PRED[State], waddr: Address, s: State):
       transformer_invariant?(states,
                              memory_write_transformers(pm′,
                                                        address_block(waddr, length(null))))
         ∧ states(s)
         ⊃ result_pred(states)(expr_2_super(memory_write_list_nse(pm′)(waddr, null))(s))
```

Expanding the definition of memory_write_list_nse,
Expanding the definition(s) of (ok_result expr_2_super expr_2_super_res result_pred has_next_state),
Repeatedly Skolemizing and flattening,
This completes the proof of `transformer_invariant_write_list_nse.1`.
`transformer_invariant_write_list_nse.2:`

```
{1}  ∀ (cons1_var: Byte, cons2_var: list[Byte]):
       (∀ (states: PRED[State], waddr: Address, s: State):
          transformer_invariant?(states,
                                 memory_write_transformers(pm′,
                                                           address_block(waddr,
                                                                         length(cons2_
          ∧ states(s)
          ⊃
          result_pred(states)
                      (expr_2_super(memory_write_list_nse(pm′)(waddr, cons2_var))(s)))
        ⊃
        (∀ (states: PRED[State], waddr: Address, s: State):
          transformer_invariant?(states,
                                 memory_write_transformers(pm′,
                                                           address_block(waddr,
                                                                         length
                                                                         (cons
                                                                           (cons1_va
                                                                             cons2_v
          ∧ states(s)
          ⊃
          result_pred(states)
                      (expr_2_super(memory_write_list_nse(pm′)
                                    (waddr, cons(cons1_var, cons2_
                                    (s)))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_list_nse,
Installing automatic rewrites from: length super_comp_expr_forget_expr! address_block
Using lemma memory_write_transformers_memory_write[State],
Simplifying, rewriting, and recording with decision procedures,
Case splitting on OK?(expr_2_super(memory_write(pm!1)(waddr!1, cons1_var!1))(s!1)),
we get 2 subgoals:

`transformer_invariant_write_list_nse.2.1:`

```
{-1}  OK?(expr_2_super(memory_write(pm')(waddr', cons1_var'))(s'))
{-2}  memory_write_transformers(pm', address_block(waddr', 1 + length(cons2_var')))
                                  (expr_2_super(memory_write(pm')(waddr', cons1_var')))
{-3}  ∀ (states: PRED[State], waddr: Address, s: State):
        transformer_invariant?(states,
                               memory_write_transformers(pm',
                                                         address_block(waddr,
                                                                       length(cons2_var'))))
        ∧ states(s)
        ⊃
        result_pred(states)
                   (expr_2_super(memory_write_list_nse(pm')(waddr, cons2_var'))(s))
{-4}  transformer_invariant?(states',
                             memory_write_transformers(pm',
                                                       address_block(waddr',
                                                                     1 + length(cons2_var'))))
{-5}  states'(s')
------------------------------------------------------------------------------------------
 {1}  result_pred(states')
                 (CASES expr_2_super(memory_write(pm')(waddr', cons1_var'))(s') OF
                    OK(state):
                      expr_2_super(memory_write_list_nse(pm')(waddr' + 1, cons2_var'))
                                  (state)
                    ELSE expr_2_super(memory_write(pm')(waddr', cons1_var'))(s')
                    ENDCASES)
```

Simplifying, rewriting, and recording with decision procedures,

Rewriting using -3, matching in *,

we get 2 subgoals:

`transformer_invariant_write_list_nse.2.1.1:`

```
{-1}  OK?(expr_2_super(memory_write(pm′)(waddr′, cons1_var′))(s′))
{-2}  memory_write_transformers(pm′, address_block(waddr′, 1 + length(cons2_var′)))
                                  (expr_2_super(memory_write(pm′)(waddr′, cons1_var′)))
{-3}  ∀ (states: PRED[State], waddr: Address, s: State):
        transformer_invariant?(states,
                               memory_write_transformers(pm′,
                                                         address_block(waddr,
                                                                       length(cons2_var′
        ∧ states(s)
        ⊃
        result_pred(states)
                    (expr_2_super(memory_write_list_nse(pm′)(waddr, cons2_var′))(s))
{-4}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_va
{-5}  states′(s′)
------------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′ + 1,
                                                                     length(cons2_var′)))
{2}   result_pred(states′)
                (expr_2_super(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′))
                             (state(expr_2_super(memory_write(pm′)(waddr′, cons1_var′))
                                                 (s′))))
```

Keeping (-4 1) and hiding *,

Installing automatic rewrites from: address_block_subset_1 memory_write_transformers_mono

Using lemma transformer_invariant_mono_transformers[State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `transformer_invariant_write_list_nse.2.1.1`.

`transformer_invariant_write_list_nse.2.1.2:`

```
{-1}  OK?(expr_2_super(memory_write(pm′)(waddr′, cons1_var′))(s′))
{-2}  memory_write_transformers(pm′, address_block(waddr′, 1 + length(cons2_var′)))
                                    (expr_2_super(memory_write(pm′)(waddr′, cons1_var′)))
{-3}  ∀ (states: PRED[State], waddr: Address, s: State):
        transformer_invariant?(states,
                               memory_write_transformers(pm′,
                                                         address_block(waddr,
                                                                       length(cons2_var′))))
          ∧ states(s)
          ⊃
          result_pred(states)
                     (expr_2_super(memory_write_list_nse(pm′)(waddr, cons2_var′))(s))
{-4}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_var′))))
{-5}  states′(s′)
-------------------------------------------------------------------------------
{1}   states′(state(expr_2_super(memory_write(pm′)(waddr′, cons1_var′))(s′)))
{2}   result_pred(states′)
                 (expr_2_super(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′))
                               (state(expr_2_super(memory_write(pm′)(waddr′, cons1_var′))
                                                  (s′))))
```

Using lemma super_transformer_invariant_next_ok[State],
Installing automatic rewrites from: has_next_state
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformer_invariant_write_list_nse.2.1.2`.
`transformer_invariant_write_list_nse.2.2:`

```
{-1}  memory_write_transformers(pm′, address_block(waddr′, 1 + length(cons2_var′)))
                                    (expr_2_super(memory_write(pm′)(waddr′, cons1_var′)))
{-2}  ∀ (states: PRED[State], waddr: Address, s: State):
        transformer_invariant?(states,
                               memory_write_transformers(pm′,
                                                         address_block(waddr,
                                                                       length(cons2_var′))))
          ∧ states(s)
          ⊃
          result_pred(states)
                     (expr_2_super(memory_write_list_nse(pm′)(waddr, cons2_var′))(s))
{-3}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_var′))))
{-4}  states′(s′)
-------------------------------------------------------------------------------
{1}   OK?(expr_2_super(memory_write(pm′)(waddr′, cons1_var′))(s′))
{2}   result_pred(states′)
                 (CASES expr_2_super(memory_write(pm′)(waddr′, cons1_var′))(s′) OF
                    OK(state):
                      expr_2_super(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′))
                                  (state)
                    ELSE expr_2_super(memory_write(pm′)(waddr′, cons1_var′))(s′)
                    ENDCASES)
```

Hiding formulas: -2,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of transformer_invariant?,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformer_invariant_write_list_nse.2.2`.
Q.E.D.

C.123.6 Memory_Change_2.transformer_invariant_write_list

Terse proof for `transformer_invariant_write_list`.
`transformer_invariant_write_list:`

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
        s: State):
       transformer_invariant?(states,
                              (memory_write_transformers(pm, address_block(waddr, length(bl))
        ∧
        side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                      memory_write_side_effect(pm))
         ∧ states(s)
        ⊃ result_pred(states)(expr_2_super(memory_write_list(pm)(waddr, bl))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_list,
Rewriting using super_comp_expr_expr, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`transformer_invariant_write_list.1:`

```
{-1}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-2}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))
{-3}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-4}  states′(s′)
{-5}  OK?(expr_2_super(memory_write_side_effect(pm′)(waddr′, bl′, FALSE))(s′))
------------------------------------------------------------------------------
{1}   result_pred(states′)
            (expr_2_super(memory_write_list_nse(pm′)
                                                (waddr′,
                                                  data(memory_write_side_effect(pm′)
                                                                (waddr′, bl′, FALSE)(s′)))
                          (state(expr_2_super(memory_write_side_effect(pm′)
                                                (waddr′, bl′, FALSE))
                                              (s′))))
```

Using lemma side_effect_content_unchanged_content[State],
Installing automatic rewrites from: subset_equal ok_expr_2_super

Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Rewriting using transformer_invariant_write_list_nse, matching in *,
we get 2 subgoals:
`transformer_invariant_write_list.1.1:`

{-1} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-2} transformer_invariant?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ {expr_2
{-3} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-4} states′(s′)
{-5} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))

{1} transformer_invariant?(states′,
memory_write_transformers(pm′,
ad-
dress_block(waddr′, length(bl′))))
{2} result_pred(states′)
(expr_2_super(memory_write_list_nse(pm′)(waddr′, bl′))
(state(expr_2_super(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE))
(s′))))

Keeping (-2 1) and hiding *,
Rewriting using add_as_union, matching in *,
Using lemma transformer_invariant_mono_transformers,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using union_subset1, matching in *,
This completes the proof of `transformer_invariant_write_list.1.1`.
`transformer_invariant_write_list.1.2:`

{-1} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-2} transformer_invariant?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ {expr_2
{-3} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-4} states′(s′)
{-5} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))

{1} states′(state(expr_2_super(memory_write_side_effect(pm′)(waddr′, bl′, FALSE))(s′)))
{2} result_pred(states′)
(expr_2_super(memory_write_list_nse(pm′)(waddr′, bl′))
(state(expr_2_super(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE))
(s′))))

Hiding formulas: -1, -3, 2,

Using lemma super_transformer_invariant_next_ok[State],
Installing automatic rewrites from: has_next_state add
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformer_invariant_write_list.1.2`.
`transformer_invariant_write_list.2`:

```
{-1}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
        (bl′)
{-2}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))
{-3}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-4}  states′(s′)
------------------------------------------------------------------------------
{1}   OK?(expr_2_super(memory_write_side_effect(pm′)(waddr′, bl′, FALSE))(s′))
{2}   result_pred(states′)
                 (expr_2_super(memory_write_side_effect(pm′)(waddr′, bl′, FALSE))(s′))
```

Hiding formulas: -1, -3,
Expanding the definition of transformer_invariant?,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of add,
which is trivially true.
This completes the proof of `transformer_invariant_write_list.2`.
Q.E.D.

C.123.7 Memory_Change_2.changed_memory_invariant?_TCC1

Terse proof for `changed_memory_invariant?_TCC1`.
`changed_memory_invariant?_TCC1`:

```
------------------------------------------------------------------------------
{1}   ∀ (pm: Memory_struct[State], states: PRED[State], addresses: PRED[Address]):
        transformer_invariant?(states, memory_write_transformers(pm, addresses)) ∧
         transformer_invariant?(states, memory_read_transformers(pm, addresses))
         ⊃
         (∀ (s: State, a: Address, b: Byte):
              OK?(memory_write(pm)(a, b)(s)) ∧ addresses(a) ∧ states(s) ⊃
               OK?[State, Unit](memory_write(pm)(a, b)(s)) ∨
                Exception?[State, Unit](memory_write(pm)(a, b)(s)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `changed_memory_invariant?_TCC1`.
Q.E.D.

C.123.8 Memory_Change_2.changed_memory_invariant_mono

Terse proof for `changed_memory_invariant_mono`.

`changed_memory_invariant_mono:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State],
addresses_1, addresses_2: PRED[Address]):
(addresses_1 ⊆ addresses_2) ∧ changed_memory_invariant?(pm, states, addresses_2) ⊃
changed_memory_invariant?(pm, states, addresses_1)

Repeatedly Skolemizing and flattening,

Expanding the definition of changed_memory_invariant?,

Applying disjunctive simplification to flatten sequent,

Applying propositional simplification,

we get 3 subgoals:

`changed_memory_invariant_mono.1:`

{-1} (addresses′ ⊆ addresses″)
{-2} transformer_invariant?(states′, memory_read_transformers(pm′, addresses″))
{-3} transformer_invariant?(states′, memory_write_transformers(pm′, addresses″))
{-4} ∀ (s: State, a: Address, b: Byte):
states′(s) ∧
addresses″(a) ∧
OK?(memory_write(pm′)(a, b)(s)) ∧
OK?(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s))))
⊃ data(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s)))) = b
---
{1} transformer_invariant?(states′, memory_read_transformers(pm′, addresses′))

Hiding formulas: -3, -4,

Installing automatic rewrites from: memory_read_transformers_mono

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `changed_memory_invariant_mono.1`.

`changed_memory_invariant_mono.2:`

{-1} (addresses′ ⊆ addresses″)
{-2} transformer_invariant?(states′, memory_read_transformers(pm′, addresses″))
{-3} transformer_invariant?(states′, memory_write_transformers(pm′, addresses″))
{-4} ∀ (s: State, a: Address, b: Byte):
states′(s) ∧
addresses″(a) ∧
OK?(memory_write(pm′)(a, b)(s)) ∧
OK?(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s))))
⊃ data(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s)))) = b
---
{1} transformer_invariant?(states′, memory_write_transformers(pm′, addresses′))

Hiding formulas: -2, -4,

Installing automatic rewrites from: memory_write_transformers_mono

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `changed_memory_invariant_mono.2`.

changed_memory_invariant_mono.3:

```
{-1}  (addresses′ ⊆ addresses″)
{-2}  transformer_invariant?(states′, memory_read_transformers(pm′, addresses″))
{-3}  transformer_invariant?(states′, memory_write_transformers(pm′, addresses″))
{-4}  ∀ (s: State, a: Address, b: Byte):
        states′(s) ∧
         addresses″(a) ∧
          OK?(memory_write(pm′)(a, b)(s)) ∧
           OK?(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s))))
         ⊃ data(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s)))) = b
-------------------------------------------------------------------------
{1}   ∀ (s: State, a: Address, b: Byte):
        states′(s) ∧
         addresses′(a) ∧
          OK?(memory_write(pm′)(a, b)(s)) ∧
           OK?(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s))))
         ⊃ data(memory_read(pm′)(a)(state(memory_write(pm′)(a, b)(s)))) = b
```

Hiding formulas: -2, -3,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of changed_memory_invariant_mono.3.
Q.E.D.

C.123.9 Memory_Change_2.memory_read_list_nse_ok

Terse proof for memory_read_list_nse_ok.
memory_read_list_nse_ok:

```
-------------------------------------------------------------------------
{1}   ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Ad-
      dress, size: nat, s: State):
        transformer_invariant?(states, memory_read_transformers(pm, ad-
      dress_block(raddr, size))) ∧
         transformers_ok?(states, memory_read_transformers(pm, ad-
      dress_block(raddr, size))) ∧
          states(s)
        ⊃ OK?(memory_read_list_nse(pm)(raddr, size)(s))
```

For the top quantifier in 1, we introduce Skolem constants: (pm′ states′ _ _ _),
Inducting on size on formula 1,
we get 2 subgoals:
memory_read_list_nse_ok.1:

```
-------------------------------------------------------------------------
{1}   ∀ (raddr: Address, s: State):
        transformer_invariant?(states′, memory_read_transformers(pm′, ad-
      dress_block(raddr, 0)))
         ∧
         transformers_ok?(states′, memory_read_transformers(pm′, ad-
      dress_block(raddr, 0))) ∧
          states′(s)
        ⊃ OK?(memory_read_list_nse(pm′)(raddr, 0)(s))
```

Expanding the definition of memory_read_list_nse,
Expanding the definition of ok_result,

which is trivially true.

This completes the proof of `memory_read_list_nse_ok.1`.

`memory_read_list_nse_ok.2`:

{1} ∀ j:
(∀ (raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr, j)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j)))
∧ states′(s)
⊃ OK?(memory_read_list_nse(pm′)(raddr, j)(s)))
⊃
(∀ (raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′,
address_block(raddr, $j+1$)))
∧
transformers_ok?(states′,
memory_read_transformers(pm′, address_block(raddr, $j+1$)))
∧ states′(s)
⊃ OK?(memory_read_list_nse(pm′)(raddr, $j+1$)(s)))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: memory_read_transformers_memory_read address_block

Expanding the definition of memory_read_list_nse,

Using lemma expr_transformers_ok_ok[State, Byte],

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of ##,

Instantiating quantified variables,

Applying propositional simplification,

we get 4 subgoals:

`memory_read_list_nse_ok.2.1:`

```
{-1}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                  (state(memory_read(pm′)(raddr′)(s′))))
{-2}  OK?(memory_read(pm′)(raddr′)(s′))
{-3}  j′ ≥ 0
{-4}  transformer_invariant?(states′,
                               memory_read_transformers(pm′, ad-
      dress_block(raddr′, 1 + j′)))
{-5}  transformers_ok?(states′,
                           memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-6}  states′(s′)
-------------------------------------------------------------------------------
{1}   OK?(CASES memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                  (state(memory_read(pm′)(raddr′)(s′))) OF
            OK(state_1, data_2):
              ok_result(cons(data(memory_read(pm′)(raddr′)(s′)), data_2))(state_1),
            Exception(ex_type, state_1): Exception(ex_type, state_1),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of `memory_read_list_nse_ok.2.1`.

`memory_read_list_nse_ok.2.2:`

```
{-1}  OK?(memory_read(pm′)(raddr′)(s′))
{-2}  j′ ≥ 0
{-3}  transformer_invariant?(states′,
                               memory_read_transformers(pm′, ad-
      dress_block(raddr′, 1 + j′)))
{-4}  transformers_ok?(states′,
                           memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-5}  states′(s′)
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                               memory_read_transformers(pm′, ad-
      dress_block(raddr′ + 1, j′)))
{2}   OK?(CASES memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                  (state(memory_read(pm′)(raddr′)(s′))) OF
            OK(state_1, data_2):
              ok_result(cons(data(memory_read(pm′)(raddr′)(s′)), data_2))(state_1),
            Exception(ex_type, state_1): Exception(ex_type, state_1),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Keeping (-3 1) and hiding *,

Installing automatic rewrites from: memory_read_transformers_mono address_block_subset_1

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_nse_ok.2.2`.

`memory_read_list_nse_ok.2.3:`

```
{-1}  OK?(memory_read(pm')(raddr')(s'))
{-2}  j' ≥ 0
{-3}  transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
      dress_block(raddr', 1 + j')))
{-4}  transformers_ok?(states',
                       memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}  states'(s')
|-----------------------------------------------------------------------------
{1}   transformers_ok?(states',
                       memory_read_transformers(pm', address_block(raddr' + 1, j')))
{2}   OK?(CASES memory_read_list_nse(pm')(raddr' + 1, j')
                                          (state(memory_read(pm')(raddr')(s'))) OF
            OK(state_1, data_2):
              ok_result(cons(data(memory_read(pm')(raddr')(s')), data_2))(state_1),
            Exception(ex_type, state_1): Exception(ex_type, state_1),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Keeping (-4 1) and hiding *,
Installing automatic rewrites from: memory_read_transformers_mono address_block_subset_1
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `memory_read_list_nse_ok.2.3`.
`memory_read_list_nse_ok.2.4:`

```
{-1}  OK?(memory_read(pm')(raddr')(s'))
{-2}  j' ≥ 0
{-3}  transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
      dress_block(raddr', 1 + j')))
{-4}  transformers_ok?(states',
                       memory_read_transformers(pm', address_block(raddr', 1 + j')))
{-5}  states'(s')
|-----------------------------------------------------------------------------
{1}   states'(state(memory_read(pm')(raddr')(s')))
{2}   OK?(CASES memory_read_list_nse(pm')(raddr' + 1, j')
                                          (state(memory_read(pm')(raddr')(s'))) OF
            OK(state_1, data_2):
              ok_result(cons(data(memory_read(pm')(raddr')(s')), data_2))(state_1),
            Exception(ex_type, state_1): Exception(ex_type, state_1),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Hiding formulas: 2,
Using lemma expr_transformer_invariant_next_ok[State, Byte],
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of has_next_state,
which is trivially true.
This completes the proof of `memory_read_list_nse_ok.2.4`.
Q.E.D.

C.123.10 Memory_Change_2.memory_read_list_nse_next_length_TCC1

Terse proof for `memory_read_list_nse_next_length_TCC1`.

`memory_read_list_nse_next_length_TCC1:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Address, size: nat, s: State):
transformer_invariant?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
states(s)
⊃
OK?[State, list[Byte]](memory_read_list_nse[State](pm)(raddr, size)(s)) ∨
Exception?[State, list[Byte]](memory_read_list_nse[State](pm)(raddr, size)(s))

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_nse_ok,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `memory_read_list_nse_next_length_TCC1`.
Q.E.D.

C.123.11 Memory_Change_2.memory_read_list_nse_next_length_TCC2

Terse proof for `memory_read_list_nse_next_length_TCC2`.

`memory_read_list_nse_next_length_TCC2:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Address, size: nat, s: State):
states(state(memory_read_list_nse(pm)(raddr, size)(s))) ∧
transformer_invariant?(states, memory_read_transformers(pm, address_block(raddr, size)))
∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
states(s)
⊃ OK?[State, list[Byte]](memory_read_list_nse[State](pm)(raddr, size)(s))

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_nse_ok,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `memory_read_list_nse_next_length_TCC2`.
Q.E.D.

C.123.12 Memory_Change_2.memory_read_list_nse_next_length

Terse proof for `memory_read_list_nse_next_length`.

`memory_read_list_nse_next_length:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Address, size: nat, s: State):
transformer_invariant?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
states(s)
⊃
states(state(memory_read_list_nse(pm)(raddr, size)(s))) ∧
length(data(memory_read_list_nse(pm)(raddr, size)(s))) = size

For the top quantifier in 1, we introduce Skolem constants: (pm′ states′ _ _ _),

Inducting on size on formula 1,

we get 4 subgoals:

`memory_read_list_nse_next_length.1:`

{1} ∀ (raddr: Address, s: State):
transformer_invariant?(states′, memory_read_transformers(pm′, address_block(raddr, 0)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, 0))) ∧
states′(s)
⊃
states′(state(memory_read_list_nse(pm′)(raddr, 0)(s))) ∧
length(data(memory_read_list_nse(pm′)(raddr, 0)(s))) = 0

Expanding the definition of memory_read_list_nse,

Repeatedly Skolemizing and flattening,

Expanding the definition of ok_result,

Expanding the definition of length,

which is trivially true.

This completes the proof of `memory_read_list_nse_next_length.1`.

`memory_read_list_nse_next_length.2:`

{1} ∀ j:
(∀ (raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr, j)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j)))
∧ states′(s)
⊃
states′(state(memory_read_list_nse(pm′)(raddr, j)(s))) ∧
length(data(memory_read_list_nse(pm′)(raddr, j)(s))) = j)
⊃
(∀ (raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′,
address_block(raddr, $j+1$)))
∧
transformers_ok?(states′,
memory_read_transformers(pm′, address_block(raddr, $j+1$)))
∧ states′(s)
⊃
states′(state(memory_read_list_nse(pm′)(raddr, $j+1$)(s))) ∧
length(data(memory_read_list_nse(pm′)(raddr, $j+1$)(s))) =
$j+1$)

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: memory_read_transformers_memory_read address_block

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Instantiating the top quantifier in -4 with the terms: raddr′ + 1, state(memory_read(pm′)(raddr′)(s′)),

Splitting conjunctions,

we get 4 subgoals:

`memory_read_list_nse_next_length.2.1:`

{-1} $\text{states}'(\text{state}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}' + 1,\ j')(\text{state}(\text{memory\_read}(\text{pm}')(\text{raddr}')(s')))))$
$\wedge$
$\text{length}(\text{data}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}' + 1,\ j')(\text{state}(\text{memory\_read}(\text{pm}')(\text{raddr}')(s'))))) = j'$
{-2} $\text{OK?}(\text{memory\_read}(\text{pm}')(\text{raddr}')(s'))$
{-3} $\text{OK?}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}',\ 1 + j')(s'))$
{-4} $j' \geq 0$
{-5} $\text{transformer\_invariant?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{address\_block}(\text{raddr}',\ 1 + j')))$
{-6} $\text{transformers\_ok?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{address\_block}(\text{raddr}',\ 1 + j')))$
{-7} $\text{states}'(s')$

{1} $\text{states}'(\text{state}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}',\ 1 + j')(s'))) \wedge \text{length}(\text{data}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}',\ 1 + j')(s'))) = 1 + j'$

Applying disjunctive simplification to flatten sequent,

Installing automatic rewrites from: length ##! ok_result

Expanding the definition of memory_read_list_nse,

Expanding the definition of memory_read_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_read_list_nse_next_length.2.1`.

`memory_read_list_nse_next_length.2.2:`

{-1} $\text{OK?}(\text{memory\_read}(\text{pm}')(\text{raddr}')(s'))$
{-2} $\text{OK?}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}',\ 1 + j')(s'))$
{-3} $j' \geq 0$
{-4} $\text{transformer\_invariant?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{address\_block}(\text{raddr}',\ 1 + j')))$
{-5} $\text{transformers\_ok?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{address\_block}(\text{raddr}',\ 1 + j')))$
{-6} $\text{states}'(s')$

{1} $\text{transformer\_invariant?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{address\_block}(\text{raddr}' + 1,\ j')))$
{2} $\text{states}'(\text{state}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}',\ 1 + j')(s'))) \wedge \text{length}(\text{data}(\text{memory\_read\_list\_nse}(\text{pm}')(\text{raddr}',\ 1 + j')(s'))) = 1 + j'$

Installing automatic rewrites from: address_block_subset_1 memory_read_transformers_mono

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_nse_next_length.2.2`.

`memory_read_list_nse_next_length.2.3:`

```
{-1}  OK?(memory_read(pm′)(raddr′)(s′))
{-2}  OK?(memory_read_list_nse(pm′)(raddr′, 1 + j′)(s′))
{-3}  j′ ≥ 0
{-4}  transformer_invariant?(states′,
                             memory_read_transformers(pm′, ad-
      dress_block(raddr′, 1 + j′)))
{-5}  transformers_ok?(states′,
                             memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-6}  states′(s′)
------------------------------------------------------------------------
{1}   transformers_ok?(states′,
                             memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
{2}   states′(state(memory_read_list_nse(pm′)(raddr′, 1 + j′)(s′))) ∧
       length(data(memory_read_list_nse(pm′)(raddr′, 1 + j′)(s′))) = 1 + j′
```

Installing automatic rewrites from: address_block_subset_1 memory_read_transformers_mono

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_nse_next_length.2.3`.

`memory_read_list_nse_next_length.2.4:`

```
{-1}  OK?(memory_read(pm′)(raddr′)(s′))
{-2}  OK?(memory_read_list_nse(pm′)(raddr′, 1 + j′)(s′))
{-3}  j′ ≥ 0
{-4}  transformer_invariant?(states′,
                             memory_read_transformers(pm′, ad-
      dress_block(raddr′, 1 + j′)))
{-5}  transformers_ok?(states′,
                             memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-6}  states′(s′)
------------------------------------------------------------------------
{1}   states′(state(memory_read(pm′)(raddr′)(s′)))
{2}   states′(state(memory_read_list_nse(pm′)(raddr′, 1 + j′)(s′))) ∧
       length(data(memory_read_list_nse(pm′)(raddr′, 1 + j′)(s′))) = 1 + j′
```

Hiding formulas: 2,

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of has_next_state,

which is trivially true.

This completes the proof of `memory_read_list_nse_next_length.2.4`.

`memory_read_list_nse_next_length.3`:

{1} ∀ (size: nat, raddr: Address, s: State):
states′(state(memory_read_list_nse(pm′)(raddr, size)(s))) ∧
transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr, size)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size))) ∧
states′(s)
⊃ OK?[State, list[Byte]](memory_read_list_nse[State](pm′)(raddr, size)(s))

{2} ∀ (raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr, size′)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size′)))
∧ states′(s)
⊃
states′(state(memory_read_list_nse(pm′)(raddr, size′)(s))) ∧
length(data(memory_read_list_nse(pm′)(raddr, size′)(s))) = size′

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_nse_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_nse_next_length.3`.

memory_read_list_nse_next_length.4:

{1} ∀ (size: nat, raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr, size)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size))) ∧
states′(s)
⊃
OK?[State, list[Byte]](memory_read_list_nse[State](pm′)(raddr, size)(s)) ∨
Exception?[State, list[Byte]](memory_read_list_nse[State](pm′)(raddr, size)(s))

{2} ∀ (raddr: Address, s: State):
transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr, size′)))
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size′)))
∧ states′(s)
⊃
states′(state(memory_read_list_nse(pm′)(raddr, size′)(s))) ∧
length(data(memory_read_list_nse(pm′)(raddr, size′)(s))) = size′

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_nse_ok,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of memory_read_list_nse_next_length.4.
Q.E.D.

C.123.13 Memory_Change_2.memory_read_list_ok

Terse proof for memory_read_list_ok.

memory_read_list_ok:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Address, size: nat, s: State):
transformer_invariant?(states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ mem
∧
transformers_ok?(states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re
∧ states(s)
⊃ OK?(memory_read_list(pm)(raddr, size)(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: union_subset1
Expanding the definition of memory_read_list,
Using lemma memory_read_list_nse_ok,
Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`memory_read_list_ok.1:`

{-1} states′(s′)
{-2} states′(state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))
{-3} length(data(memory_read_list_nse(pm′)(raddr′, size′)(s′))) = size′
{-4} OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-5} size′ ≥ 0
{-6} transformer_invariant?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_si
{-7} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_side_effec
---
{1} OK?((memory_read_list_nse(pm′)(raddr′, size′) ##
(λ (bl1: list[Byte]): memory_read_side_effect(pm′)(raddr′, bl1, FALSE)))
(s′))

Expanding the definition of ##,

Hiding formulas: -2, -5, -6,

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -4, 2,

Installing automatic rewrites from: union_right memory_read_side_effect_super_transformers_memory_read_side_effect subset_equal

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_ok.1`.

`memory_read_list_ok.2:`

{-1} states′(s′)
{-2} size′ ≥ 0
{-3} transformer_invariant?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_si
{-4} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_side_effec
---
{1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, size′)))
{2} OK?((memory_read_list_nse(pm′)(raddr′, size′) ##
(λ (bl1: list[Byte]): memory_read_side_effect(pm′)(raddr′, bl1, FALSE)))
(s′))

Hiding formulas: 2,

Keeping (-4 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_ok.2`.

memory_read_list_ok.3:

```
{-1}  states′(s′)
{-2}  size′ ≥ 0
{-3}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ mem
{-4}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_rea
------------------------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_read_transformers(pm′, ad-
      dress_block(raddr′, size′)))
{2}   OK?((memory_read_list_nse(pm′)(raddr′, size′) ##
           (λ (bl1: list[Byte]): memory_read_side_effect(pm′)(raddr′, bl1, FALSE)))
            (s′))
```

Keeping (-3 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_ok.3.

Q.E.D.

C.123.14 Memory_Change_2.memory_read_list_next_ok_TCC1

Terse proof for memory_read_list_next_ok_TCC1.

memory_read_list_next_ok_TCC1:

```
------------------------------------------------------------------------------------------------
{1}   ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Ad-
      dress, size: nat, s: State):
        transformer_invariant?(states,
                               (memory_read_transformers(pm, address_block(raddr, size)) ∪ mem
        ∧
        transformers_ok?(states,
                         (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re
         ∧ states(s)
        ⊃
        OK?[State, list[Byte]](memory_read_list[State](pm)(raddr, size)(s)) ∨
         Exception?[State, list[Byte]](memory_read_list[State](pm)(raddr, size)(s))
```

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

Using lemma memory_read_list_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of memory_read_list_next_ok_TCC1.

Q.E.D.

C.123.15 Memory_Change_2.memory_read_list_next_ok

Terse proof for memory_read_list_next_ok.

`memory_read_list_next_ok:`

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], raddr: Ad-
     dress, size: nat, s: State):
       transformer_invariant?(states,
                              (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_read_si
       ∧
       transformers_ok?(states,
                        (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_read_side_effe
        ∧ states(s)
       ⊃ states(state(memory_read_list(pm)(raddr, size)(s)))
```

Repeatedly Skolemizing and flattening,

Using lemma memory_read_list_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`memory_read_list_next_ok.1:`

```
{-1}  states′(s′)
{-2}  states′(state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))
{-3}  length(data(memory_read_list_nse(pm′)(raddr′, size′)(s′))) = size′
{-4}  OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-5}  OK?(memory_read_side_effect(pm′)
                 (raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
                 (state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
{-6}  size′ ≥ 0
{-7}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_si
{-8}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_side_effe
-------------------------------------------------------------------------------------------------------------
{1}  states′(state(memory_read_side_effect(pm′)
                   (raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)),
                    FALSE)
                   (state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))))
```

Hiding formulas: -1, -6, -8,

Installing automatic rewrites from: has_next_state union_right memory_read_side_effect_super_transformers_memory_read_si subset_equal

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `memory_read_list_next_ok.1`.

`memory_read_list_next_ok.2:`

{-1} states′(s′)
{-2} OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-3} OK?(memory_read_side_effect(pm′)
(raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
{-4} size′ ≥ 0
{-5} transformer_invariant?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ mem
{-6} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_rea
---
{1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, size′)))
{2} states′(state(memory_read_side_effect(pm′)
(raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)),
FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))))

Keeping (-6 1) and hiding *,
Installing automatic rewrites from: union_subset1
Using lemma transformers_ok_mono_transformers,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `memory_read_list_next_ok.2`.
`memory_read_list_next_ok.3:`

{-1} states′(s′)
{-2} OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-3} OK?(memory_read_side_effect(pm′)
(raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
{-4} size′ ≥ 0
{-5} transformer_invariant?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ mem
{-6} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_rea
---
{1} transformer_invariant?(states′,
memory_read_transformers(pm′, address_block(raddr′, size′)))
{2} states′(state(memory_read_side_effect(pm′)
(raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)),
FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))))

Keeping (-5 1) and hiding *,
Installing automatic rewrites from: union_subset1
Using lemma transformer_invariant_mono_transformers,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `memory_read_list_next_ok.3`.
Q.E.D.

C.123.16 Memory_Change_2.unchanged_memory_read_list_nse_TCC1

Terse proof for `unchanged_memory_read_list_nse_TCC1`.

unchanged_memory_read_list_nse_TCC1:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
size: nat, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ singleton(
addresses)
∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
(address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?($q(s)$)
⊃ OK?[State]($q(s)$) ∨ abnormal?[State]($q(s)$)

Repeatedly Skolemizing and flattening,

This completes the proof of unchanged_memory_read_list_nse_TCC1.

Q.E.D.

C.123.17 Memory_Change_2.unchanged_memory_read_list_nse_TCC2

Terse proof for unchanged_memory_read_list_nse_TCC2.

unchanged_memory_read_list_nse_TCC2:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
size: nat, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ singleton(
addresses)
∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
(address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?($q(s)$)
⊃
OK?[State, list[Byte]]
(memory_read_list_nse[State](pm)(raddr, size)(state[State]($q(s)$))))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: has_next_state union_right singleton union_subset1

Using lemma memory_read_list_nse_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_read_list_nse_TCC2.1`:

```
{-1}  transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}  size' ≥ 0
{-3}  unchanged_memory_invariant?(pm', states',
                                  (memory_read_transformers(pm', address_block(raddr', size')) ∪
                                  addresses')
{-4}  (address_block(raddr', size') ⊆ addresses')
{-5}  states'(s')
{-6}  OK?(q'(s'))
------------------------------------------------------------------------------
{1}   states'(state[State](q'(s')))
{2}   OK?[State, list[Byte]]
          (memory_read_list_nse[State](pm')(raddr', size')(state[State](q'(s'))))
```

Using lemma unchanged_memory_invariant_next_ok[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse_TCC2.1`.

`unchanged_memory_read_list_nse_TCC2.2`:

```
{-1}  transformers_ok?(states', memory_read_transformers(pm', address_block(raddr', size')))
{-2}  size' ≥ 0
{-3}  unchanged_memory_invariant?(pm', states',
                                  (memory_read_transformers(pm', address_block(raddr', size')) ∪
                                  addresses')
{-4}  (address_block(raddr', size') ⊆ addresses')
{-5}  states'(s')
{-6}  OK?(q'(s'))
------------------------------------------------------------------------------
{1}   transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
      dress_block(raddr', size')))
{2}   OK?[State, list[Byte]]
          (memory_read_list_nse[State](pm')(raddr', size')(state[State](q'(s'))))
```

Forward chaining on unchanged_memory_invariant_invariant,

Keeping (-1 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_nse_TCC2.2`.

Q.E.D.

C.123.18 Memory_Change_2.unchanged_memory_read_list_nse_TCC3

Terse proof for `unchanged_memory_read_list_nse_TCC3`.

`unchanged_memory_read_list_nse_TCC3`:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
size: nat, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ singleton(
addresses)
∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
(address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?(q(s))
⊃ OK?[State, list[Byte]](memory_read_list_nse[State](pm)(raddr, size)(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: union_subset1
Using lemma memory_read_list_nse_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Forward chaining on unchanged_memory_invariant_invariant,
Keeping (-1 1) and hiding *,
Using lemma transformer_invariant_mono_transformers,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unchanged_memory_read_list_nse_TCC3`.
Q.E.D.

C.123.19 Memory_Change_2.unchanged_memory_read_list_nse

Terse proof for `unchanged_memory_read_list_nse`.

`unchanged_memory_read_list_nse`:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
size: nat, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ singleton(
addresses)
∧
transformers_ok?(states, memory_read_transformers(pm, address_block(raddr, size))) ∧
(address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm)(raddr, size)(state(q(s)))) =
data(memory_read_list_nse(pm)(raddr, size)(s))

For the top quantifier in 1, we introduce Skolem constants: (addresses′ pm′ states′ _ _ _ _),
Inducting on size on formula 1,
we get 5 subgoals:

`unchanged_memory_read_list_nse.1`:

```
{1}  ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
       unchanged_memory_invariant?(pm′, states′,
                                   (memory_read_transformers(pm′, address_block(raddr, 0)) ∪ s
                                   addresses′)
         ∧
         transformers_ok?(states′, memory_read_transformers(pm′, ad-
       dress_block(raddr, 0))) ∧
          (address_block(raddr, 0) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
         ⊃
         data(memory_read_list_nse(pm′)(raddr, 0)(state(q(s)))) =
          data(memory_read_list_nse(pm′)(raddr, 0)(s))
```

Expanding the definition of memory_read_list_nse,
Expanding the definition of ok_result,
which is trivially true.
This completes the proof of `unchanged_memory_read_list_nse.1`.
`unchanged_memory_read_list_nse.2`:

```
{1}  ∀ j:
       (∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
            unchanged_memory_invariant?(pm′, states′,
                                        (memory_read_transformers(pm′, address_block(raddr, j
                                        addresses′)
              ∧
              transformers_ok?(states′, memory_read_transformers(pm′, ad-
       dress_block(raddr, j)))
               ∧ (address_block(raddr, j) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
              ⊃
              data(memory_read_list_nse(pm′)(raddr, j)(state(q(s)))) =
               data(memory_read_list_nse(pm′)(raddr, j)(s)))
         ⊃
         (∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
            unchanged_memory_invariant?(pm′, states′,
                                        (memory_read_transformers(pm′, address_block(raddr,
                                        addresses′)
              ∧
              transformers_ok?(states′,
                                  memory_read_transformers(pm′, ad-
       dress_block(raddr, j + 1)))
               ∧
               (address_block(raddr, j + 1) ⊆ addresses′) ∧
                states′(s) ∧ OK?(q(s))
              ⊃
              data(memory_read_list_nse(pm′)(raddr, j + 1)(state(q(s)))) =
               data(memory_read_list_nse(pm′)(raddr, j + 1)(s)))
```

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_read_list_nse_TCC2,
Using lemma unchanged_memory_read_list_nse_TCC3,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list_nse,

Expanding the definition of memory_read_list_nse,

Expanding the definition of memory_read_list_nse,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -3, -6, -7,

Expanding the definition of ok_result,

Applying extensionality,

we get 2 subgoals:

`unchanged_memory_read_list_nse.2.1:`

{-1} OK?(memory_read(pm′)(raddr′)(s′))
{-2} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)(raddr′)(s′))))
{-3} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-4} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)
(raddr′)(state[State](q′(s′))))))
{-5} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, j′)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, ad-
dress_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-6} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr′, 1 + j′)) ∪ singleto
addresses′)
{-7} transformers_ok?(states′,
memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-8} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-9} states′(s′)
{-10} OK?(q′(s′))

{1} data(memory_read(pm′)(raddr′)(state(q′(s′)))) =
data(memory_read(pm′)(raddr′)(s′))

Hiding formulas: -2, -4, -5, -7,

Using lemma unchanged_memory_invariant_unchanged[State],

Installing automatic rewrites from: union_right singleton

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-6 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.1`.

unchanged_memory_read_list_nse.2.2:

{-1} OK?(memory_read(pm′)(raddr′)(s′))
{-2} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)(raddr′)(s′))))
{-3} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-4} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)
(raddr′)(state[State](q′(s′))))))
{-5} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, j′))
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-6} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr′, 1 + j′
addresses′)
{-7} transformers_ok?(states′,
memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-8} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-9} states′(s′)
{-10} OK?(q′(s′))

{1} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)(raddr′)(state(q′(s′))))))
=
data(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)(raddr′)(s′))))

Installing automatic rewrites from: state_expr_2_super has_next_state address_block_subset_2 ok_expr_2_super subset_equal union_subset1 union_subset3 unchanged_memory_invariant_union_transformers union_right singleton memory_read_transformers_mono

Case splitting on unchanged_memory_invariant?(pm!1, states!1, memory_read_transformers(pm!1, address_block (raddr!1 + 1, j!1)), addresses!1),

we get 2 subgoals:

`unchanged_memory_read_list_nse.2.2.1:`

{-1} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)), addresses′)
{-2} OK?(memory_read(pm′)(raddr′)(s′))
{-3} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))
{-4} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-5} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state[State](q′(s′))))))
{-6} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr, j′)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-7} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr′, 1 + j′)) ∪ singleto
addresses′)
{-8} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-9} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-10} states′(s′)
{-11} OK?(q′(s′))

{1} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′))))))
=
data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))

Case splitting on unchanged_memory_invariant?(pm!1, states!1, singleton(expr_2_super (memory_read(pm!1)(raddr!1))), addresses!1),

we get 2 subgoals:

`unchanged_memory_read_list_nse.2.2.1.1:`

{-1} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_read(pm′)(raddr′))), addresses′)
{-2} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)), addresses′)
{-3} OK?(memory_read(pm′)(raddr′)(s′))
{-4} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))
{-5} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-6} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state[State](q′(s′))))))
{-7} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]): unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr, j′)) addresses′) ∧ transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧ (address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s)) ⊃ data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) = data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-8} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr′, 1 + j′ addresses′)
{-9} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-10} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-11} states′(s′)
{-12} OK?(q′(s′))

{1} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′)))))) = data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))

Case splitting on transformers_ok?(states!1, memory_read_transformers(pm!1, address_block (raddr!1 + 1, j!1))),

we get 2 subgoals:

`unchanged_memory_read_list_nse.2.2.1.1.1:`

{-1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
{-2} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_read(pm′)(raddr′))), addresses′)
{-3} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)), addresses′)
{-4} OK?(memory_read(pm′)(raddr′)(s′))
{-5} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))
{-6} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-7} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state[State](q′(s′))))))
{-8} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr, j′)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-9} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr′, 1 + j′)) ∪ singleto
addresses′)
{-10} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-11} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-12} states′(s′)
{-13} OK?(q′(s′))

{1} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′))))))
=
data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))

Copying formula number: -8
Instantiating the top quantifier in -1 with the terms: raddr′ + 1, s′, expr_2_super(memory_read(pm′)(raddr′)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -3,
Hiding formulas: -3,
Copying formula number: -9
Instantiating the top quantifier in -1 with the terms: raddr′ + 1, state(q′(s′)), expr_2_super(memory_read(pm′)(raddr′)),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`unchanged_memory_read_list_nse.2.2.1.1.1.1:`

{-1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
{-2} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′)))))) = data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(q′(s′))))
{-3} states′(s′)
{-4} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_read(pm′)(raddr′))), addresses′)
{-5} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)), addresses′)
{-6} OK?(memory_read(pm′)(raddr′)(s′))
{-7} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))
{-8} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-9} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state[State](q′(s′))))))
{-10} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]): unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr, j′)) addresses′) ∧ transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧ (address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s)) ⊃ data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) = data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-11} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr′, 1 + j′ addresses′)
{-12} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-13} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-14} OK?(q′(s′))

{1} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′)))))) = data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(s′))

Replacing using formula -2,

Hiding formulas: -2,

Case splitting on unchanged_memory_invariant?(pm!1, states!1, singleton(q!1), addresses!1),

we get 2 subgoals:

`unchanged_memory_read_list_nse.2.2.1.1.1.1.1:`

{-1} unchanged_memory_invariant?(pm′, states′, singleton(q′), addresses′)
{-2} transformers_ok?(states′,
memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
{-3} states′(s′)
{-4} unchanged_memory_invariant?(pm′, states′,
singleton(expr_2_super(memory_read(pm′)(raddr′))),
addresses′)
{-5} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(raddr′ + 1, j′)),
addresses′)
{-6} OK?(memory_read(pm′)(raddr′)(s′))
{-7} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)(raddr′)(s′))))
{-8} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-9} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
(state(memory_read(pm′)
(raddr′)(state[State](q′(s′))))))
{-10} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, j′)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-11} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr′, 1 + j′)) ∪ singleto
addresses′)
{-12} transformers_ok?(states′,
memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-13} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-14} OK?(q′(s′))

{1} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(q′(s′)))) =
data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(s′))

Rewriting using -10, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_nse.2.2.1.1.1.1.1`.

`unchanged_memory_read_list_nse.2.2.1.1.1.1.2:`

```
{-1}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
{-2}  states′(s′)
{-3}  unchanged_memory_invariant?(pm′, states′,
                                  singleton(expr_2_super(memory_read(pm′)(raddr′))),
                                  addresses′)
{-4}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(raddr′ + 1, j′)),
                                  addresses′)
{-5}  OK?(memory_read(pm′)(raddr′)(s′))
{-6}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                  (state(memory_read(pm′)(raddr′)(s′))))
{-7}  OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-8}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                  (state(memory_read(pm′)
                                                    (raddr′)(state[State](q′(s′))))))
{-9}  ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
        unchanged_memory_invariant?(pm′, states′,
                                    (memory_read_transformers(pm′, address_block(raddr, j′))
                                    addresses′)
          ∧
          transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
            (address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
          ⊃
          data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
            data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-10} unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(raddr′, 1 + j′
                                  addresses′)
{-11} transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-12} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-13} OK?(q′(s′))
------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm′, states′, singleton(q′), addresses′)
 {2}  data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(q′(s′)))) =
        data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(s′))
```

Keeping (-10 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.2.1.1.1.1.2`.

`unchanged_memory_read_list_nse.2.2.1.1.1.2:`

{-1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
{-2} states′(s′)
{-3} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_read(pm′)(raddr′))), addresses′)
{-4} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)), addresses′)
{-5} OK?(memory_read(pm′)(raddr′)(s′))
{-6} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))
{-7} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-8} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state[State](q′(s′))))))
{-9} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr, j′)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-10} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr′, 1 + j′)) ∪ singleto
addresses′)
{-11} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-12} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-13} OK?(q′(s′))

{1} states′(state(q′(s′)))
{2} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′))))))
= data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(s′))

Keeping (-10 1) and hiding *,

Using lemma unchanged_memory_invariant_next_ok[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.2.1.1.1.2`.

`unchanged_memory_read_list_nse.2.2.1.1.2:`

```
{-1}  unchanged_memory_invariant?(pm′, states′,
                                  singleton(expr_2_super(memory_read(pm′)(raddr′))),
                                  addresses′)
{-2}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(raddr′ + 1, j′)),
                                  addresses′)
{-3}  OK?(memory_read(pm′)(raddr′)(s′))
{-4}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)(raddr′)(s′))))
{-5}  OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-6}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)
                                                  (raddr′)(state[State](q′(s′))))))
{-7}  ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
        unchanged_memory_invariant?(pm′, states′,
                                    (memory_read_transformers(pm′, address_block(raddr, j′))
                                    addresses′)
          ∧
          transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
           (address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
          ⊃
          data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
           data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-8}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(raddr′, 1 + j′
                                  addresses′)
{-9}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-10} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-11} states′(s′)
{-12} OK?(q′(s′))
-------------------------------------------------------------------------------
 {1}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(raddr′ + 1, j′)))
 {2}  data(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)(raddr′)(state(q′(s′))))))
       =
      data(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)(raddr′)(s′))))
```

Keeping (-9 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.2.1.1.2`.

`unchanged_memory_read_list_nse.2.2.1.2:`

{-1} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(raddr′ + 1, j′)), addresses′)
{-2} OK?(memory_read(pm′)(raddr′)(s′))
{-3} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))
{-4} OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-5} OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state[State](q′(s′))))))
{-6} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]): unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr, j′)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, j′))) ∧
(address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
⊃
data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-7} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(raddr′, 1 + j′)) ∪ singleto
addresses′)
{-8} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-9} (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-10} states′(s′)
{-11} OK?(q′(s′))

{1} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(memory_read(pm′)(raddr′))), addresses′)
{2} data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(state(q′(s′)))))) =
data(memory_read_list_nse(pm′)(raddr′ + 1, j′)(state(memory_read(pm′)(raddr′)(s′))))

Keeping (-7 1) and hiding *,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: -1,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using singleton_subset, matching in *,
Expanding the definition of member,
Rewriting using memory_read_transformers_memory_read, matching in *,
Keeping 1 and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_read_list_nse.2.2.1.2`.

`unchanged_memory_read_list_nse.2.2.2:`

```
{-1}  OK?(memory_read(pm′)(raddr′)(s′))
{-2}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)(raddr′)(s′))))
{-3}  OK?(memory_read(pm′)(raddr′)(state[State](q′(s′))))
{-4}  OK?(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)
                                                  (raddr′)(state[State](q′(s′))))))
{-5}  ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
        unchanged_memory_invariant?(pm′, states′,
                                    (memory_read_transformers(pm′, address_block(raddr, j′))
                                    addresses′)
          ∧
          transformers_ok?(states′, memory_read_transformers(pm′, ad-
      dress_block(raddr, j′))) ∧
           (address_block(raddr, j′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
          ⊃
          data(memory_read_list_nse(pm′)(raddr, j′)(state(q(s)))) =
           data(memory_read_list_nse(pm′)(raddr, j′)(s))
{-6}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(raddr′, 1 + j′
                                  addresses′)
{-7}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(raddr′, 1 + j′)))
{-8}  (address_block(raddr′, 1 + j′) ⊆ addresses′)
{-9}  states′(s′)
{-10} OK?(q′(s′))
-------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           ad-
      dress_block(raddr′ + 1, j′)),
                                  addresses′)
 {2}  data(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)(raddr′)(state(q′(s′))))))
       =
       data(memory_read_list_nse(pm′)(raddr′ + 1, j′)
                                (state(memory_read(pm′)(raddr′)(s′))))
```

Keeping (-6 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_nse.2.2.2`.

`unchanged_memory_read_list_nse.3`:

{1} ∀ (size: nat, raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, size)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size))) ∧
(address_block(raddr, size) ⊆ addresses′) ∧ states′(s) ∧ OK?($q(s)$)
⊃ OK?[State, list[Byte]](memory_read_list_nse[State](pm′)(raddr, size)(s))

{2} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, size′)) ∪ singleton
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size′)))
∧ (address_block(raddr, size′) ⊆ addresses′) ∧ states′(s) ∧ OK?($q(s)$)
⊃
data(memory_read_list_nse(pm′)(raddr, size′)(state($q(s)$))) =
data(memory_read_list_nse(pm′)(raddr, size′)(s))

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_read_list_nse_TCC3,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_read_list_nse.3`.
`unchanged_memory_read_list_nse.4`:

{1} ∀ (size: nat, raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, size)) ∪ singleton(
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size))) ∧
(address_block(raddr, size) ⊆ addresses′) ∧ states′(s) ∧ OK?($q(s)$)
⊃
OK?[State, list[Byte]]
(memory_read_list_nse[State](pm′)(raddr, size)(state[State]($q(s)$))))

{2} ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(raddr, size′)) ∪ singleton
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr, size′)))
∧ (address_block(raddr, size′) ⊆ addresses′) ∧ states′(s) ∧ OK?($q(s)$)
⊃
data(memory_read_list_nse(pm′)(raddr, size′)(state($q(s)$))) =
data(memory_read_list_nse(pm′)(raddr, size′)(s))

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_read_list_nse_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_read_list_nse.4`.
`unchanged_memory_read_list_nse.5`:

```
{1}  ∀ (size: nat, raddr: Address, s: State, q: [State → SuperResult[State]]):
       unchanged_memory_invariant?(pm′, states′,
                                   (memory_read_transformers(pm′, address_block(raddr, size))
                                   addresses′)
        ∧
        transformers_ok?(states′, memory_read_transformers(pm′, ad-
     dress_block(raddr, size))) ∧
         (address_block(raddr, size) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
        ⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))
{2}  ∀ (raddr: Address, s: State, q: [State → SuperResult[State]]):
       unchanged_memory_invariant?(pm′, states′,
                                   (memory_read_transformers(pm′, address_block(raddr, size′))
                                   addresses′)
        ∧
        transformers_ok?(states′, memory_read_transformers(pm′, ad-
     dress_block(raddr, size′)))
         ∧ (address_block(raddr, size′) ⊆ addresses′) ∧ states′(s) ∧ OK?(q(s))
        ⊃
        data(memory_read_list_nse(pm′)(raddr, size′)(state(q(s)))) =
         data(memory_read_list_nse(pm′)(raddr, size′)(s))
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
This completes the proof of `unchanged_memory_read_list_nse.5`.
Q.E.D.

C.123.20 Memory_Change_2.unchanged_memory_read_list_TCC1

Terse proof for `unchanged_memory_read_list_TCC1`.
`unchanged_memory_read_list_TCC1`:

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Ad-
     dress,
          size: nat, s: State, q: [State → SuperResult[State]]):
       unchanged_memory_invariant?(pm, states,
                                   ((memory_read_transformers(pm, address_block(raddr, size))
                                   addresses)
        ∧
        transformers_ok?(states,
                         (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re
         ∧
         side_effect_content_unchanged(address_block(raddr, size), states,
                                       memory_read_side_effect(pm))
          ∧ (address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?(q(s))
        ⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))
```

Repeatedly Skolemizing and flattening,

This completes the proof of `unchanged_memory_read_list_TCC1`.

Q.E.D.

C.123.21 Memory_Change_2.unchanged_memory_read_list_TCC2

Terse proof for `unchanged_memory_read_list_TCC2`.

`unchanged_memory_read_list_TCC2`:

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
        size: nat, s: State, q: [State → SuperResult[State]]):
      unchanged_memory_invariant?(pm, states,
                                  ((memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_
                                  addresses)
       ∧
       transformers_ok?(states,
                        (memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_read_side_effe
        ∧
        side_effect_content_unchanged(address_block(raddr, size), states,
                                      memory_read_side_effect(pm))
         ∧ (address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?(q(s))
       ⊃
       OK?[State, list[Byte]]
          (memory_read_list[State](pm)(raddr, size)(state[State](q(s))))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: union singleton member union_subset1 has_next_state

Using lemma unchanged_memory_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma memory_read_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-4 1) and hiding *,

Forward chaining on unchanged_memory_invariant_invariant,

Hiding formulas: -2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_TCC2`.

Q.E.D.

C.123.22 Memory_Change_2.unchanged_memory_read_list_TCC3

Terse proof for `unchanged_memory_read_list_TCC3`.

unchanged_memory_read_list_TCC3:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
size: nat, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states,
((memory_read_transformers(pm, address_block(raddr, size))
addresses)
∧
transformers_ok?(states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re
∧
side_effect_content_unchanged(address_block(raddr, size), states,
memory_read_side_effect(pm))
∧ (address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?(q(s))
⊃ OK?[State, list[Byte]](memory_read_list[State](pm)(raddr, size)(s))

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-4 1) and hiding *,
Forward chaining on unchanged_memory_invariant_invariant,
Hiding formulas: -2,
Using lemma transformer_invariant_mono_transformers,
Installing automatic rewrites from: union_subset1
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of unchanged_memory_read_list_TCC3.
Q.E.D.

C.123.23 Memory_Change_2.unchanged_memory_read_list

Terse proof for unchanged_memory_read_list.

unchanged_memory_read_list:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], raddr: Address,
size: nat, s: State, q: [State → SuperResult[State]]):
unchanged_memory_invariant?(pm, states,
((memory_read_transformers(pm, address_block(raddr, size))
addresses)
∧
transformers_ok?(states,
(memory_read_transformers(pm, address_block(raddr, size)) ∪ memory_re
∧
side_effect_content_unchanged(address_block(raddr, size), states,
memory_read_side_effect(pm))
∧ (address_block(raddr, size) ⊆ addresses) ∧ states(s) ∧ OK?(q(s))
⊃
data(memory_read_list(pm)(raddr, size)(state(q(s)))) =
data(memory_read_list(pm)(raddr, size)(s))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 has_next_state
Using lemma unchanged_memory_read_list_TCC2,
Using lemma unchanged_memory_read_list_TCC3,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of memory_read_list,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma memory_read_list_nse_next_length,
Using lemma memory_read_list_nse_next_length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`unchanged_memory_read_list.1:`

{-1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, size′)))
{-2} states′(s′)
{-3} states′(state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))
{-4} length(data(memory_read_list_nse(pm′)(raddr′, size′)(s′))) = size′
{-5} states′(state(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′)))))
{-6} length(data(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′))))) = size′
{-7} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_side_effe
{-8} OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-9} OK?[State, list[Byte]]
(memory_read_side_effect(pm′)
(raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
{-10} OK?(memory_read_list_nse(pm′)(raddr′, size′)(state[State](q′(s′))))
{-11} OK?[State, list[Byte]]
(memory_read_side_effect(pm′)
(raddr′,
data(memory_read_list_nse(pm′)(raddr′, size′)
(state[State](q′(s′)))),
FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)
(state[State](q′(s′))))))
{-12} size′ ≥ 0
{-13} unchanged_memory_invariant?(pm′, states′,
((memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory
addresses′)
{-14} side_effect_content_unchanged(address_block(raddr′, size′), states′,
memory_read_side_effect(pm′))
{-15} (address_block(raddr′, size′) ⊆ addresses′)
{-16} OK?(q′(s′))

{1} data(memory_read_side_effect(pm′)
(raddr′,
data(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′)))), FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′))))))
=
data(memory_read_side_effect(pm′)
(raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
(state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))

Using lemma side_effect_content_unchanged_content [State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Hiding formulas: -4,

Using lemma side_effect_content_unchanged_content [State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -4,

Hiding formulas: -4,

Using lemma unchanged_memory_read_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-16 1) and hiding *,

Rewriting using union_commutative, matching in -1,

Rewriting using union_associative, matching in *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list.1`.

`unchanged_memory_read_list.2:`

```
{-1}  transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, size′)))
{-2}  states′(s′)
{-3}  states′(state(memory_read_list_nse(pm′)(raddr′, size′)(s′)))
{-4}  length(data(memory_read_list_nse(pm′)(raddr′, size′)(s′))) = size′
{-5}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_read_side_effe
{-6}  OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-7}  OK?[State, list[Byte]]
        (memory_read_side_effect(pm′)
              (raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
              (state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
{-8}  OK?(memory_read_list_nse(pm′)(raddr′, size′)(state[State](q′(s′))))
{-9}  OK?[State, list[Byte]]
        (memory_read_side_effect(pm′)
              (raddr′,
                data(memory_read_list_nse(pm′)(raddr′, size′)
                                              (state[State](q′(s′)))),
                FALSE)
              (state(memory_read_list_nse(pm′)(raddr′, size′)
                                               (state[State](q′(s′))))))
{-10} size′ ≥ 0
{-11} unchanged_memory_invariant?(pm′, states′,
                                  ((memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory
                                  addresses′)
{-12} side_effect_content_unchanged(address_block(raddr′, size′), states′,
                                    memory_read_side_effect(pm′))
{-13} (address_block(raddr′, size′) ⊆ addresses′)
{-14} OK?(q′(s′))
-------------------------------------------------------------------------------
 {1}  states′(state(q′(s′)))
 {2}  data(memory_read_side_effect(pm′)
              (raddr′,
                data(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′)))), FALSE)
              (state(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′))))))
      =
      data(memory_read_side_effect(pm′)
                (raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
                (state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
```

Keeping (-11 1) and hiding *,

Installing automatic rewrites from: union member singleton

Using lemma unchanged_memory_invariant_next_ok[State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list.2`.

`unchanged_memory_read_list.3:`

```
{-1}  transformers_ok?(states′, memory_read_transformers(pm′, address_block(raddr′, size′)))
{-2}  states′(s′)
{-3}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(raddr′, size′)) ∪ memory_re
{-4}  OK?(memory_read_list_nse(pm′)(raddr′, size′)(s′))
{-5}  OK?[State, list[Byte]]
          (memory_read_side_effect(pm′)
                (raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
                (state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
{-6}  OK?(memory_read_list_nse(pm′)(raddr′, size′)(state[State](q′(s′))))
{-7}  OK?[State, list[Byte]]
          (memory_read_side_effect(pm′)
                (raddr′,
                  data(memory_read_list_nse(pm′)(raddr′, size′)
                                                (state[State](q′(s′)))),
                  FALSE)
                (state(memory_read_list_nse(pm′)(raddr′, size′)
                                                (state[State](q′(s′))))))
{-8}  size′ ≥ 0
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  ((memory_read_transformers(pm′, address_block(raddr′, size′)
                                  addresses′)
{-10} side_effect_content_unchanged(address_block(raddr′, size′), states′,
                                    memory_read_side_effect(pm′))
{-11} (address_block(raddr′, size′) ⊆ addresses′)
{-12} OK?(q′(s′))
-------------------------------------------------------------------------------
 {1}  transformer_invariant?(states′,
                             memory_read_transformers(pm′, ad-
      dress_block(raddr′, size′)))
 {2}  data(memory_read_side_effect(pm′)
                (raddr′,
                  data(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′)))), FALSE)
                (state(memory_read_list_nse(pm′)(raddr′, size′)(state(q′(s′))))))
       =
      data(memory_read_side_effect(pm′)
                 (raddr′, data(memory_read_list_nse(pm′)(raddr′, size′)(s′)), FALSE)
                 (state(memory_read_list_nse(pm′)(raddr′, size′)(s′))))
```

Keeping (-9 1) and hiding *,

Forward chaining on unchanged_memory_invariant_invariant,

Hiding formulas: -2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list.3`.

Q.E.D.

C.123.24 Memory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_read_list_nse_TCC1`.

`unchanged_memory_invariant_unchanged_read_list_nse_TCC1`:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], s: State, addr_1, addr_2: Address, size: nat):
unchanged_memory_invariant?(pm, states, memory_read_transformers(pm, address_block(addr_2, size)), addresses)
∧
transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧
(address_block(addr_2, size) ⊆ addresses) ∧ addresses(addr_1) ∧ states(s)
⊃
OK?[State, list[Byte]](memory_read_list_nse[State](pm)(addr_2, size)(s)) ∨
Exception?[State, list[Byte]](memory_read_list_nse[State](pm)(addr_2, size)(s))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: memory_read_transformers_mono

Using lemma memory_read_list_nse_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_invariant_unchanged_read_list_nse_TCC1.1`:

{-1} states′(s′)
{-2} size′ ≥ 0
{-3} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(addr″, size′)), addresses′)
{-4} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5} (address_block(addr″, size′) ⊆ addresses′)
{-6} addresses′(addr′)

{1} transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr″, size′)))
{2} OK?[State, list[Byte]](memory_read_list_nse[State](pm′)(addr″, size′)(s′))
{3} Exception?[State, list[Byte]](memory_read_list_nse[State](pm′)(addr″, size′)(s′))

Keeping (-3 -4 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse_TCC1.1`.

unchanged_memory_invariant_unchanged_read_list_nse_TCC1.2:

```
{-1}  states′(s′)
{-2}  size′ ≥ 0
{-3}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′, ad-
      dress_block(addr″, size′)),
                                  addresses′)
{-4}  transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5}  (address_block(addr″, size′) ⊆ addresses′)
{-6}  addresses′(addr′)
------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_read_transformers(pm′, ad-
      dress_block(addr″, size′)))
{2}   OK?[State, list[Byte]](memory_read_list_nse[State](pm′)(addr″, size′)(s′))
{3}   Exception?[State, list[Byte]](memory_read_list_nse[State](pm′)(addr″, size′)(s′))
```

Forward chaining on unchanged_memory_invariant_invariant,
This completes the proof of unchanged_memory_invariant_unchanged_read_list_nse_TCC1.2.
Q.E.D.

C.123.25 Memory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse_TCC2

Terse proof for unchanged_memory_invariant_unchanged_read_list_nse_TCC2.

unchanged_memory_invariant_unchanged_read_list_nse_TCC2:

```
------------------------------------------------------------------------
{1}   ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], s: State,
         addr_1, addr_2: Address, size: nat):
        unchanged_memory_invariant?(pm, states,
                                    memory_read_transformers(pm, ad-
      dress_block(addr_2, size)),
                                    addresses)
          ∧
          transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧
           (address_block(addr_2, size) ⊆ addresses) ∧ addresses(addr_1) ∧ states(s)
          ⊃
          OK?[State, Byte]
              (memory_read(pm)
                          (addr_1)
                          (state[State, list[Byte]]
                                 (memory_read_list_nse[State](pm)(addr_2, size)(s))))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: memory_read_transformers_memory_read
Using lemma expr_transformers_ok_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: 2,
Using lemma memory_read_list_nse_next_length,
Forward chaining on unchanged_memory_invariant_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Installing automatic rewrites from: memory_read_transformers_mono
Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse_TCC2`.
Q.E.D.

C.123.26 Memory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse_TCC3

Terse proof for `unchanged_memory_invariant_unchanged_read_list_nse_TCC3`.
`unchanged_memory_invariant_unchanged_read_list_nse_TCC3`:

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], s: State,
          addr_1, addr_2: Address, size: nat):
       unchanged_memory_invariant?(pm, states,
                                   memory_read_transformers(pm, ad-
    dress_block(addr_2, size)),
                                   addresses)
         ∧
         transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧
          (address_block(addr_2, size) ⊆ addresses) ∧ addresses(addr_1) ∧ states(s)
         ⊃ OK?[State, Byte](memory_read(pm)(addr_1)(s))
```

Repeatedly Skolemizing and flattening,
Using lemma expr_transformers_ok_ok[State, Byte],
Installing automatic rewrites from: memory_read_transformers_memory_read
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse_TCC3`.
Q.E.D.

C.123.27 Memory_Change_2.unchanged_memory_invariant_unchanged_read_list_nse

Terse proof for `unchanged_memory_invariant_unchanged_read_list_nse`.
`unchanged_memory_invariant_unchanged_read_list_nse`:

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], s: State,
          addr_1, addr_2: Address, size: nat):
       unchanged_memory_invariant?(pm, states,
                                   memory_read_transformers(pm, ad-
    dress_block(addr_2, size)),
                                   addresses)
         ∧
         transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧
          (address_block(addr_2, size) ⊆ addresses) ∧ addresses(addr_1) ∧ states(s)
         ⊃
         data(memory_read(pm)(addr_1)(state(memory_read_list_nse(pm)(addr_2, size)(s))))
          = data(memory_read(pm)(addr_1)(s))
```

For the top quantifier in 1, we introduce Skolem constants: (addresses′ pm′ states′ _ addr′ _ _),
Inducting on size on formula 1,
we get 5 subgoals:

`unchanged_memory_invariant_unchanged_read_list_nse.1:`

```
{1}  ∀ (s: State, addr_2: Address):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′, ad-
     dress_block(addr_2, 0)),
                                   addresses′)
         ∧
         transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
          (address_block(addr_2, 0) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
         ⊃
         data(memory_read(pm′)
                         (addr′)(state(memory_read_list_nse(pm′)(addr_2, 0)(s))))
          = data(memory_read(pm′)(addr′)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.1`.
`unchanged_memory_invariant_unchanged_read_list_nse.2:`

```
{1}  ∀ j:
       (∀ (s: State, addr_2: Address):
            unchanged_memory_invariant?(pm′, states′,
                                        memory_read_transformers(pm′, ad-
     dress_block(addr_2, j)),
                                        addresses′)
              ∧
              transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
               (address_block(addr_2, j) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
              ⊃
              data(memory_read(pm′)
                              (addr′)(state(memory_read_list_nse(pm′)(addr_2, j)(s))))
               = data(memory_read(pm′)(addr′)(s)))
         ⊃
         (∀ (s: State, addr_2: Address):
              unchanged_memory_invariant?(pm′, states′,
                                          memory_read_transformers(pm′,
                                                                   address_block
                                                                   (addr_2, j + 1)),
                                          addresses′)
               ∧
               transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
                (address_block(addr_2, j + 1) ⊆ addresses′) ∧
                 addresses′(addr′) ∧ states′(s)
               ⊃
               data(memory_read(pm′)
                               (addr′)
                               (state(memory_read_list_nse(pm′)(addr_2, j + 1)(s))))
                = data(memory_read(pm′)(addr′)(s)))
```

Repeatedly Skolemizing and flattening,
Forward chaining on unchanged_memory_invariant_invariant,
Installing automatic rewrites from: memory_read_transformers_mono ok_result address_block_subset_1

address_block_subset_2 address_block has_next_state memory_read_transformers_memory_read subset_equal

Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma memory_read_list_nse_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of memory_read_list_nse,
Expanding the definition of memory_read_list_nse,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: -6,
Using lemma expr_transformer_invariant_next_ok[State, Byte],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`unchanged_memory_invariant_unchanged_read_list_nse.2.1:`

{-1} transformer_invariant?(states′, memory_read_transformers(pm′, address_block(addr″, 1 + j′)))
{-2} states′(s′)
{-3} states′(state(memory_read(pm′)(addr″)(s′)))
{-4} transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr″, 1 + j′)))
{-5} OK?(memory_read(pm′)(addr″)(s′))
{-6} OK?(memory_read_list_nse(pm′)(addr″ + 1, j′)(state(memory_read(pm′)(addr″)(s′))))
{-7} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-8} j′ ≥ 0
{-9} data(memory_read(pm′)(addr′)(state(memory_read_list_nse(pm′)(addr″ + 1, j′)(state(memory_read(pm′)(addr″)(s′)))))) = data(memory_read(pm′)(addr′)(state(memory_read(pm′)(addr″)(s′))))
{-10} unchanged_memory_invariant?(pm′, states′, memory_read_transformers(pm′, address_block(addr″, 1 + j′)), addresses′)
{-11} (address_block(addr″, 1 + j′) ⊆ addresses′)
{-12} addresses′(addr′)

{1} data(memory_read(pm′)(addr′)(state(memory_read_list_nse(pm′)(addr″ + 1, j′)(state(memory_read(pm′)(addr″)(s′)))))) = data(memory_read(pm′)(addr′)(s′))

Replacing using formula -9,
Hiding formulas: -9,
Using lemma expr_unchanged_memory_invariant_unchanged[State, Byte],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`unchanged_memory_invariant_unchanged_read_list_nse.2.1.1:`

```
{-1}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(addr″, 1 + j′)),
                                  addresses′)
{-2}  states′(s′)
{-3}  addresses′(addr′)
{-4}  OK?(memory_read(pm′)(addr″)(s′))
{-5}  transformer_invariant?(states′,
                             memory_read_transformers(pm′, address_block(addr″, 1 + j′)))
{-6}  states′(state(memory_read(pm′)(addr″)(s′)))
{-7}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(addr″, 1 + j′)))
{-8}  OK?(memory_read_list_nse(pm′)(addr″ + 1, j′)
                            (state(memory_read(pm′)(addr″)(s′))))
{-9}  transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-10} j′ ≥ 0
{-11} (address_block(addr″, 1 + j′) ⊆ addresses′)
-------------------------------------------------------------------------------
 {1}  OK?(memory_read(pm′)(addr′)(state(memory_read(pm′)(addr″)(s′))))
 {2}  data(memory_read(pm′)(addr′)(state(memory_read(pm′)(addr″)(s′)))) =
       data(memory_read(pm′)(addr′)(s′))
```

Keeping (-3 -6 -9 1) and hiding *,
Using lemma expr_transformers_ok_ok[State, Byte],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.2.1.1`.
`unchanged_memory_invariant_unchanged_read_list_nse.2.1.2:`

```
{-1}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(addr″, 1 + j′)),
                                  addresses′)
{-2}  states′(s′)
{-3}  addresses′(addr′)
{-4}  OK?(memory_read(pm′)(addr″)(s′))
{-5}  transformer_invariant?(states′,
                             memory_read_transformers(pm′, address_block(addr″, 1 + j′)))
{-6}  states′(state(memory_read(pm′)(addr″)(s′)))
{-7}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(addr″, 1 + j′)))
{-8}  OK?(memory_read_list_nse(pm′)(addr″ + 1, j′)
                            (state(memory_read(pm′)(addr″)(s′))))
{-9}  transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-10} j′ ≥ 0
{-11} (address_block(addr″, 1 + j′) ⊆ addresses′)
-------------------------------------------------------------------------------
 {1}  OK?(memory_read(pm′)(addr′)(s′))
 {2}  data(memory_read(pm′)(addr′)(state(memory_read(pm′)(addr″)(s′)))) =
       data(memory_read(pm′)(addr′)(s′))
```

Keeping (-3 -9 1) and hiding *,

Using lemma expr_transformers_ok_ok[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.2.1.2`.

`unchanged_memory_invariant_unchanged_read_list_nse.2.2`:

```
 {-1}  transformer_invariant?(states',
                              memory_read_transformers(pm', ad-
       dress_block(addr'', 1 + j')))
 {-2}  states'(s')
 {-3}  states'(state(memory_read(pm')(addr'')(s')))
 {-4}  transformers_ok?(states',
                        memory_read_transformers(pm', address_block(addr'', 1 + j')))
 {-5}  OK?(memory_read(pm')(addr'')(s'))
 {-6}  OK?(memory_read_list_nse(pm')(addr'' + 1, j')
                                 (state(memory_read(pm')(addr'')(s'))))
 {-7}  transformers_ok?(states', memory_read_transformers(pm', addresses'))
 {-8}  j' ≥ 0
 {-9}  unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm',
                                                            ad-
       dress_block(addr'', 1 + j')),
                                   addresses')
{-10}  (address_block(addr'', 1 + j') ⊆ addresses')
{-11}  addresses'(addr')
-------------------------------------------------------------------------------
  {1}  unchanged_memory_invariant?(pm', states',
                                   memory_read_transformers(pm',
                                                            ad-
       dress_block(addr'' + 1, j')),
                                   addresses')
  {2}  data(memory_read(pm')
            (addr')
            (state(memory_read_list_nse(pm')(addr'' + 1, j')
                                            (state(memory_read(pm')(addr'')(s'))))))
         = data(memory_read(pm')(addr')(s'))
```

Keeping (-9 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.2.2`.

`unchanged_memory_invariant_unchanged_read_list_nse.3:`

```
{1}  ∀ (size: nat, s: State, addr_2: Address):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′, ad-
     dress_block(addr_2, size)),
                                   addresses′)
         ∧
         transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
          (address_block(addr_2, size) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
         ⊃ OK?[State, Byte](memory_read(pm′)(addr′)(s))
{2}  ∀ (s: State, addr_2: Address):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′, ad-
     dress_block(addr_2, size′)),
                                   addresses′)
         ∧
         transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
          (address_block(addr_2, size′) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
         ⊃
         data(memory_read(pm′)
                         (addr′)(state(memory_read_list_nse(pm′)(addr_2, size′)(s))))
          = data(memory_read(pm′)(addr′)(s))
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_read_list_nse_TCC3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.3`.

`unchanged_memory_invariant_unchanged_read_list_nse.4:`

{1} ∀ (size: nat, s: State, addr_2: Address):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′, address_block(addr_2, size)),
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
(address_block(addr_2, size) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
⊃
OK?[State, Byte]
(memory_read(pm′)
(addr′)
(state[State, list[Byte]]
(memory_read_list_nse[State](pm′)(addr_2, size)(s))))

{2} ∀ (s: State, addr_2: Address):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′, address_block(addr_2, size′)),
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
(address_block(addr_2, size′) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
⊃
data(memory_read(pm′)
(addr′)(state(memory_read_list_nse(pm′)(addr_2, size′)(s))))
= data(memory_read(pm′)(addr′)(s))

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_read_list_nse_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.4`.

`unchanged_memory_invariant_unchanged_read_list_nse.5:`

{1} ∀ (size: nat, s: State, addr_2: Address):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′, address_block(addr_2, size)),
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
(address_block(addr_2, size) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
⊃
OK?[State, list[Byte]](memory_read_list_nse[State](pm′)(addr_2, size)(s)) ∨
Exception?[State, list[Byte]](memory_read_list_nse[State](pm′)(addr_2, size)(s))

{2} ∀ (s: State, addr_2: Address):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′, address_block(addr_2, size′)),
addresses′)
∧
transformers_ok?(states′, memory_read_transformers(pm′, addresses′)) ∧
(address_block(addr_2, size′) ⊆ addresses′) ∧ addresses′(addr′) ∧ states′(s)
⊃
data(memory_read(pm′)
(addr′)(state(memory_read_list_nse(pm′)(addr_2, size′)(s))))
= data(memory_read(pm′)(addr′)(s))

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_invariant_unchanged_read_list_nse_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_read_list_nse.5`.
Q.E.D.

C.123.28 Memory_Change_2.memory_write_list_ok_nse

Terse proof for `memory_write_list_ok_nse`.

`memory_write_list_ok_nse:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, s: State,
bl: list[Byte]):
transformer_invariant?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧ states(s)
⊃ OK?(memory_write_list_nse(pm)(waddr, bl)(s))

For the top quantifier in 1, we introduce Skolem constants: (pm′ states′ _ _ _),
Inducting on bl on formula 1,

we get 2 subgoals:

`memory_write_list_ok_nse.1:`

{1} ∀ (waddr: Address, s: State):
transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr, length(null))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(null))))
∧ states′(s)
⊃ OK?(memory_write_list_nse(pm′)(waddr, null)(s))

Expanding the definition of memory_write_list_nse,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of `memory_write_list_ok_nse.1`.

`memory_write_list_ok_nse.2:`

```
──────────────────────────────────────────────────────────────────────────────
{1}  ∀ (cons1_var: Byte, cons2_var: list[Byte]):
       (∀ (waddr: Address, s: State):
          transformer_invariant?(states′,
                                 memory_write_transformers(pm′,
                                                           address_block(waddr,
                                                                         length(cons2_
          ∧
          transformers_ok?(states′,
                           memory_write_transformers(pm′,
                                                     ad-
     dress_block(waddr, length(cons2_var))))
           ∧ states′(s)
          ⊃ OK?(memory_write_list_nse(pm′)(waddr, cons2_var)(s)))
        ⊃
       (∀ (waddr: Address, s: State):
          transformer_invariant?(states′,
                                 memory_write_transformers(pm′,
                                                           ad-
     dress_block(waddr,
                                                                       length
                                                                       (cons
                                                                          (cons1_var
                                                                             cons2_v
          ∧
          transformers_ok?(states′,
                           memory_write_transformers(pm′,
                                                     address_block(waddr,
                                                                   length
                                                                   (cons
                                                                      (cons1_var,
                                                                         cons2_var))))
           ∧ states′(s)
          ⊃ OK?(memory_write_list_nse(pm′)(waddr, cons(cons1_var, cons2_var))(s)))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: memory_write_transformers_memory_write address_block length comp_expr_forget_expr has_next_state address_block_subset_1 memory_write_transformers_mono

Expanding the definition of memory_write_list_nse,

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Simplifying, rewriting, and recording with decision procedures,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`memory_write_list_ok_nse.2.1:`

```
{-1}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_var′))))
{-3}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-4}  states′(s′)
-------------------------------------------------------------------------------
{1}   states′(state(memory_write(pm′)(waddr′, cons1_var′)(s′)))
{2}   OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
```

Using lemma expr_transformer_invariant_next_ok[State, Unit],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok_nse.2.1`.

`memory_write_list_ok_nse.2.2:`

```
{-1}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_var′))))
{-3}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-4}  states′(s′)
-------------------------------------------------------------------------------
{1}   transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′ + 1,
                                                               length(cons2_var′))))
{2}   OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
```

Keeping (-3 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok_nse.2.2`.

`memory_write_list_ok_nse.2.3:`

```
{-1}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_va
{-3}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-4}  states′(s′)
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′ + 1,
                                                                     length(cons2_var′)))
{2}   OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                   (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
```

Keeping (-2 1) and hiding *,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `memory_write_list_ok_nse.2.3`.
Q.E.D.

C.123.29 Memory_Change_2.memory_write_list_ok

Terse proof for `memory_write_list_ok`.
`memory_write_list_ok:`

```
-------------------------------------------------------------------------------
{1}   ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, s: State,
           bl: list[Byte]):
        transformer_invariant?(states,
                               (memory_write_transformers(pm, address_block(waddr, length(bl))
         ∧
         transformers_ok?(states,
                          (memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ me
          ∧
          side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                        memory_write_side_effect(pm))
           ∧ states(s)
         ⊃ OK?(memory_write_list(pm)(waddr, bl)(s))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Installing automatic rewrites from: union member has_next_state union_subset1 memory_write_side_effect_super_transformers_memory_write_side_effect subset_equal
Expanding the definition of memory_write_list,
Using lemma expr_transformers_ok_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ##,
Using lemma side_effect_content_unchanged_content[State],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_write_list_ok_nse,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`memory_write_list_ok.1:`

{-1} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-2} states′(s′)
{-3} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-4} data(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)) = bl′
{-5} transformers_ok?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write_
{-6} transformer_invariant?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory
---
{1} states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{2} OK?(memory_write_list_nse(pm′)
(waddr′,
data(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′)))
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))

Keeping (-6 1) and hiding *,
Using lemma expr_transformer_invariant_next_ok[State, list[Byte]],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `memory_write_list_ok.1`.
`memory_write_list_ok.2:`

{-1} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-2} states′(s′)
{-3} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-4} data(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)) = bl′
{-5} transformers_ok?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write_
{-6} transformer_invariant?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory
---
{1} transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
length
(data
(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)
(s′))))))
{2} OK?(memory_write_list_nse(pm′)
(waddr′,
data(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′)))
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))

Replacing using formula -4,
Keeping (-5 1) and hiding *,
Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok.2`.

`memory_write_list_ok.3`:

```
{-1}  side_effect_content_unchanged(address_block(waddr', length(bl')), states',
                                    memory_write_side_effect(pm'))
{-2}  states'(s')
{-3}  OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-4}  data(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')) = bl'
{-5}  transformers_ok?(states',
                       (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ mem
{-6}  transformer_invariant?(states',
                             (memory_write_transformers(pm', address_block(waddr', length(bl'))
-------------------------------------------------------------------------------------------------
{1}   transformer_invariant?(states',
                             memory_write_transformers(pm',
                                                       address_block(waddr',
                                                                     length
                                                                     (data
                                                                        (memory_write_si
                                                                          (pm')
                                                                          (waddr', bl',
                                                                          (s')))))))
{2}   OK?(memory_write_list_nse(pm')
                              (waddr',
                                 data(memory_write_side_effect(pm')
                                                  (waddr', bl', FALSE)(s')))
                              (state(memory_write_side_effect(pm')
                                                  (waddr', bl', FALSE)(s'))))
```

Replacing using formula -4,

Keeping (-6 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `memory_write_list_ok.3`.

Q.E.D.

C.123.30 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_write_list_nse_TCC1`.

unchanged_memory_invariant_unchanged_write_list_nse_TCC1:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
bl: list[Byte]):
unchanged_memory_invariant?(pm, states,
memory_write_transformers(pm,
address_block(waddr, length(bl))),
singleton(raddr))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
states(s)
⊃
OK?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)) ∨
Exception?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s))

Repeatedly Skolemizing and flattening,

Using lemma memory_write_list_ok_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Forward chaining on unchanged_memory_invariant_invariant,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_nse_TCC1.

Q.E.D.

C.123.31 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse_TCC2

Terse proof for unchanged_memory_invariant_unchanged_write_list_nse_TCC2.

unchanged_memory_invariant_unchanged_write_list_nse_TCC2:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
bl: list[Byte]):
unchanged_memory_invariant?(pm, states,
memory_write_transformers(pm,
address_block(waddr, length(bl))),
singleton(raddr))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
states(s)
⊃
OK?[State, Byte]
(memory_read(pm)
(raddr)
(state[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s))))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: singleton

Using lemma expr_transformers_ok_ok[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_write_list_ok_nse,

Simplifying, rewriting, and recording with decision procedures,

Using lemma transformer_invariant_write_list_nse,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of result_pred,

Installing automatic rewrites from: has_next_state_expr_2_super has_next_state state_expr_2_super

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_invariant_unchanged_write_list_nse_TCC2.

Q.E.D.

C.123.32 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse_TCC3

Terse proof for unchanged_memory_invariant_unchanged_write_list_nse_TCC3.

unchanged_memory_invariant_unchanged_write_list_nse_TCC3:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
bl: list[Byte]):
unchanged_memory_invariant?(pm, states,
memory_write_transformers(pm,
address_block(waddr, length(bl))),
singleton(raddr))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
states(s)
⊃ OK?[State, Byte](memory_read(pm)(raddr)(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: singleton
Using lemma expr_transformers_ok_ok[State, Byte],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of unchanged_memory_invariant_unchanged_write_list_nse_TCC3.
Q.E.D.

C.123.33 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_nse

Terse proof for unchanged_memory_invariant_unchanged_write_list_nse.

unchanged_memory_invariant_unchanged_write_list_nse:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
bl: list[Byte]):
unchanged_memory_invariant?(pm, states,
memory_write_transformers(pm,
address_block(waddr, length(bl))),
singleton(raddr))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
states(s)
⊃
data(memory_read(pm)(raddr)(state(memory_write_list_nse(pm)(waddr, bl)(s)))) =
data(memory_read(pm)(raddr)(s))

For the top quantifier in 1, we introduce Skolem constants: (pm′ states′ _ raddr′ _ _),

Inducting on bl on formula 1,

we get 5 subgoals:

`unchanged_memory_invariant_unchanged_write_list_nse.1:`

{1} ∀ (*s*: State, waddr: Address):
unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
address_block(waddr, length(null))),
singleton(raddr′))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(null))))
∧
transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
states′(*s*)
⊃
data(memory_read(pm′)
(raddr′)(state(memory_write_list_nse(pm′)(waddr, null)(*s*))))
= data(memory_read(pm′)(raddr′)(*s*))

Expanding the definition of memory_write_list_nse,

Expanding the definition of ok_result,

which is trivially true.

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.1`.

unchanged_memory_invariant_unchanged_write_list_nse.2:

```
{1}  ∀ (cons1_var: Byte, cons2_var: list[Byte]):
       (∀ (s: State, waddr: Address):
          unchanged_memory_invariant?(pm′, states′,
                                      memory_write_transformers(pm′,
                                                                address_block
                                                                (waddr, length(cons2_var))),
                                      singleton(raddr′))
          ∧
          transformers_ok?(states′,
                           memory_write_transformers(pm′,
                                                     address_block(waddr, length(cons2_var))))
          ∧
          transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
           ∧ states′(s)
          ⊃
          data(memory_read(pm′)
                          (raddr′)(state(memory_write_list_nse(pm′)(waddr, cons2_var)(s))))
           = data(memory_read(pm′)(raddr′)(s)))
       ⊃
       (∀ (s: State, waddr: Address):
          unchanged_memory_invariant?(pm′, states′,
                                      memory_write_transformers(pm′,
                                                                address_block
                                                                (waddr,
                                                                   length
                                                                   (cons
                                                                      (cons1_var, cons2_var)))),
                                      singleton(raddr′))
          ∧
          transformers_ok?(states′,
                           memory_write_transformers(pm′,
                                                     address_block(waddr,
                                                                   length
                                                                   (cons
                                                                      (cons1_var,
                                                                         cons2_var)))))
           ∧
           transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
            ∧ states′(s)
           ⊃
           data(memory_read(pm′)
                           (raddr′)
                           (state(memory_write_list_nse(pm′)(waddr, cons(cons1_var, cons2_var))
                                                            (s))))
            = data(memory_read(pm′)(raddr′)(s)))
```

Repeatedly Skolemizing and flattening,
Forward chaining on unchanged_memory_invariant_invariant,
Using lemma memory_write_list_ok_nse,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: memory_write_transformers_memory_write singleton address_block_subset_1 memory_write_transformers_mono length ## address_block has_next_state subset_equal

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_transformers_ok_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_write_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using -7, matching in *,

we get 3 subgoals:

`unchanged_memory_invariant_unchanged_write_list_nse.2.1:`

```
{-1}  OK?(memory_read(pm′)
                (raddr′)(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-2}  states′(state(memory_write(pm′)(waddr′, cons1_var′)(s′)))
{-3}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-4}  OK?(memory_read(pm′)(raddr′)(s′))
{-5}  OK?(memory_write_list_nse(pm′)(waddr′, cons(cons1_var′, cons2_var′))(s′))
{-6}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_var′))))
{-7}  ∀ (s: State, waddr: Address):
        unchanged_memory_invariant?(pm′, states′,
                                    memory_write_transformers(pm′,
                                                              address_block(waddr,
                                                                            length(cons2_var′))),
                                    singleton(raddr′))
          ∧
          transformers_ok?(states′,
                           memory_write_transformers(pm′,
                                                     address_block(waddr, length(cons2_var′))))
           ∧ states′(s)
          ⊃
          data(memory_read(pm′)
                          (raddr′)(state(memory_write_list_nse(pm′)(waddr, cons2_var′)(s))))
           = data(memory_read(pm′)(raddr′)(s))
{-8}  unchanged_memory_invariant?(pm′, states′,
                                  memory_write_transformers(pm′,
                                                            address_block(waddr′,
                                                                          1 + length(cons2_var′))),
                                  singleton(raddr′))
{-9}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-10} transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-11} states′(s′)
-------------------------------------------------------------------------------
 {1}  data(memory_read(pm′)
                  (raddr′)(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
       = data(memory_read(pm′)(raddr′)(s′))
```

Hiding formulas: -7,

Using lemma expr_unchanged_memory_invariant_unchanged,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.2.1`.

unchanged_memory_invariant_unchanged_write_list_nse.2.2:

{-1} OK?(memory_read(pm′)
(raddr′)(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-2} states′(state(memory_write(pm′)(waddr′, cons1_var′)(s′)))
{-3} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-4} OK?(memory_read(pm′)(raddr′)(s′))
{-5} OK?(memory_write_list_nse(pm′)(waddr′, cons(cons1_var′, cons2_var′))(s′))
{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v
{-7} ∀ (s: State, waddr: Address):
unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
ad-
dress_block(waddr,
length(con
singleton(raddr′))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr, length(cons2_var′
∧ states′(s)
⊃
data(memory_read(pm′)
(raddr′)(state(memory_write_list_nse(pm′)(waddr, cons2_var′)(s))))
= data(memory_read(pm′)(raddr′)(s))
{-8} unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(co
singleton(raddr′))
{-9} transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))))
{-10} transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-11} states′(s′)

{1} unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
ad-
dress_block(waddr′ + 1,
length(cons2
singleton(raddr′))
{2} data(memory_read(pm′)
(raddr′)
(state(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))
= data(memory_read(pm′)(raddr′)(s′))

Keeping (-8 1) and hiding *,
Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.2.2`.
`unchanged_memory_invariant_unchanged_write_list_nse.2.3`:

```
{-1}  OK?(memory_read(pm′)
                (raddr′)(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-2}  states′(state(memory_write(pm′)(waddr′, cons1_var′)(s′)))
{-3}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-4}  OK?(memory_read(pm′)(raddr′)(s′))
{-5}  OK?(memory_write_list_nse(pm′)(waddr′, cons(cons1_var′, cons2_var′))(s′))
{-6}  transformer_invariant?(states′,
                               memory_write_transformers(pm′,
                                                         address_block(waddr′,
                                                                       1 + length(cons2_var′))))
{-7}  ∀ (s: State, waddr: Address):
        unchanged_memory_invariant?(pm′, states′,
                                    memory_write_transformers(pm′,
                                                              ad-
      dress_block(waddr,
                                                                                  length(cons2_var′))),
                                    singleton(raddr′))
          ∧
          transformers_ok?(states′,
                           memory_write_transformers(pm′,
                                                     address_block(waddr, length(cons2_var′))))
           ∧ states′(s)
          ⊃
          data(memory_read(pm′)
                     (raddr′)(state(memory_write_list_nse(pm′)(waddr, cons2_var′)(s))))
           = data(memory_read(pm′)(raddr′)(s))
{-8}  unchanged_memory_invariant?(pm′, states′,
                                  memory_write_transformers(pm′,
                                                            address_block(waddr′,
                                                                          1 + length(cons2_var′)))
                                  singleton(raddr′))
{-9}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-10} transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-11} states′(s′)
-------------------------------------------------------------------------------
 {1}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′ + 1,
                                                               length(cons2_var′))))
 {2}  data(memory_read(pm′)
                 (raddr′)
                 (state(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                                  (state(memory_write(pm′)
                                                                     (waddr′, cons1_var′)(s′))))))
       = data(memory_read(pm′)(raddr′)(s′))
```

Keeping (-9 1) and hiding *,
Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.2.3`.

`unchanged_memory_invariant_unchanged_write_list_nse.3`:

```
{1}  ∀ (bl: list[Byte], s: State, waddr: Address):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_write_transformers(pm′,
                                                             address_block(waddr, length(bl))),
                                   singleton(raddr′))
         ∧
         transformers_ok?(states′,
                          memory_write_transformers(pm′, address_block(waddr, length(bl))))
          ∧
          transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
           states′(s)
         ⊃ OK?[State, Byte](memory_read(pm′)(raddr′)(s))
{2}  ∀ (s: State, waddr: Address):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_write_transformers(pm′,
                                                             address_block(waddr, length(bl′))),
                                   singleton(raddr′))
         ∧
         transformers_ok?(states′,
                          memory_write_transformers(pm′, address_block(waddr, length(bl′))))
          ∧
          transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
           states′(s)
         ⊃
         data(memory_read(pm′)
                         (raddr′)(state(memory_write_list_nse(pm′)(waddr, bl′)(s))))
          = data(memory_read(pm′)(raddr′)(s))
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_write_list_nse_TCC3,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.3`.

`unchanged_memory_invariant_unchanged_write_list_nse.4:`

{1} ∀ (bl: list[Byte], s: State, waddr: Address):
unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
address_block(waddr, length(bl))),
singleton(raddr′))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(bl))))
∧
transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
states′(s)
⊃
OK?[State, Byte]
(memory_read(pm′)
(raddr′)
(state[State, Unit](memory_write_list_nse[State](pm′)(waddr, bl)(s))))

{2} ∀ (s: State, waddr: Address):
unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
address_block(waddr, length(bl′))),
singleton(raddr′))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(bl′))))
∧
transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
states′(s)
⊃
data(memory_read(pm′)
(raddr′)(state(memory_write_list_nse(pm′)(waddr, bl′)(s))))
= data(memory_read(pm′)(raddr′)(s))

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_write_list_nse_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.4`.

`unchanged_memory_invariant_unchanged_write_list_nse.5:`

{1} ∀ (bl: list[Byte], s: State, waddr: Address):
unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
address_block(waddr, length(bl))),
singleton(raddr′))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(bl))))
∧
transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
states′(s)
⊃
OK?[State, Unit](memory_write_list_nse[State](pm′)(waddr, bl)(s)) ∨
Exception?[State, Unit](memory_write_list_nse[State](pm′)(waddr, bl)(s))

{2} ∀ (s: State, waddr: Address):
unchanged_memory_invariant?(pm′, states′,
memory_write_transformers(pm′,
address_block(waddr, length(bl′))),
singleton(raddr′))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(bl′))))
∧
transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′)))) ∧
states′(s)
⊃
data(memory_read(pm′)
(raddr′)(state(memory_write_list_nse(pm′)(waddr, bl′)(s))))
= data(memory_read(pm′)(raddr′)(s))

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_unchanged_write_list_nse_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_nse.5`.

Q.E.D.

C.123.34 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_TCC1

Terse proof for `unchanged_memory_invariant_unchanged_write_list_TCC1`.

`unchanged_memory_invariant_unchanged_write_list_TCC1:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
bl: list[Byte]):
unchanged_memory_invariant?(pm, states,
(memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ me
singleton(raddr))
∧
transformers_ok?(states,
(memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ memory_write
∧
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
memory_write_side_effect(pm))
∧ states(s)
⊃
OK?[State, Unit](memory_write_list[State](pm)(waddr, bl)(s)) ∨
Exception?[State, Unit](memory_write_list[State](pm)(waddr, bl)(s))

Repeatedly Skolemizing and flattening,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_write_list_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_TCC1`.

Q.E.D.

C.123.35 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_TCC2

Terse proof for `unchanged_memory_invariant_unchanged_write_list_TCC2`.

unchanged_memory_invariant_unchanged_write_list_TCC2:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
bl: list[Byte]):
unchanged_memory_invariant?(pm, states,
(memory_write_transformers(pm, address_block(waddr, lengt
singleton(raddr))
∧
transformers_ok?(states,
(memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ me
∧
transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
memory_write_side_effect(pm))
∧ states(s)
⊃
OK?[State, Byte]
(memory_read(pm)
(raddr)
(state[State, Unit](memory_write_list[State](pm)(waddr, bl)(s))))

Repeatedly Skolemizing and flattening,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_write_list_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_transformers_ok_ok,

Installing automatic rewrites from: singleton union_right has_next_state add_as_union memory_write_side_effect_super_transformers_memory_write_side_effect union_subset1 subset_equal subset_singleton union_right

Simplifying, rewriting, and recording with decision procedures,

Using lemma transformer_invariant_write_list,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_invariant_unchanged_write_list_TCC2.1:`

```
{-1}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-2}  states′(s′)
{-3}  result_pred(states′)(expr_2_super(memory_write_list(pm′)(waddr′, bl′))(s′))
{-4}  OK?(memory_write_list(pm′)(waddr′, bl′)(s′))
{-5}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory
{-6}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ m
                                  singleton(raddr′))
{-7}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write_
{-8}  transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
-------------------------------------------------------------------------------
 {1}  states′(state(memory_write_list[State](pm′)(waddr′, bl′)(s′)))
 {2}  OK?[State, Byte]
          (memory_read(pm′)
                  (raddr′)
                  (state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′))))
```

Rewriting using expr_result_pred_next_state, matching in *,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list_TCC2.1`.

`unchanged_memory_invariant_unchanged_write_list_TCC2.2:`

```
{-1}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-2}  states′(s′)
{-3}  OK?(memory_write_list(pm′)(waddr′, bl′)(s′))
{-4}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory
{-5}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ m
                                  singleton(raddr′))
{-6}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write_
{-7}  transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
-------------------------------------------------------------------------------
 {1}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ singleto
 {2}  states′(state(memory_write_list[State](pm′)(waddr′, bl′)(s′)))
 {3}  OK?[State, Byte]
          (memory_read(pm′)
                  (raddr′)
                  (state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′))))
```

Keeping (-4 1) and hiding *,

Rewriting using transformer_invariant_union_transformers, matching in *,

we get 2 subgoals:

unchanged_memory_invariant_unchanged_write_list_TCC2.2.1:

```
{-1}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))
{1}   transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       ad-
      dress_block(waddr′, length(bl′))))
{2}   transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))
```

Hiding formulas: 2,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of unchanged_memory_invariant_unchanged_write_list_TCC2.2.1.
unchanged_memory_invariant_unchanged_write_list_TCC2.2.2:

```
{-1}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))
{1}   transformer_invariant?(states′,
                             singleton(expr_2_super(memory_write_side_effect(pm′)
                                                    (waddr′, bl′, FALSE))))
{2}   transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))
```

Hiding formulas: 2,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of unchanged_memory_invariant_unchanged_write_list_TCC2.2.2.
Q.E.D.

C.123.36 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list_TCC3

Terse proof for unchanged_memory_invariant_unchanged_write_list_TCC3.
unchanged_memory_invariant_unchanged_write_list_TCC3:

```
{1}   ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Ad-
      dress,
          bl: list[Byte]):
        unchanged_memory_invariant?(pm, states,
                                    (memory_write_transformers(pm, address_block(waddr, lengt
                                    singleton(raddr))
          ∧
          transformers_ok?(states,
                           (memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ me
           ∧
           transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
            side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                          memory_write_side_effect(pm))
             ∧ states(s)
          ⊃ OK?[State, Byte](memory_read(pm)(raddr)(s))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: singleton
Using lemma expr_transformers_ok_ok[State, Byte],
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unchanged_memory_invariant_unchanged_write_list_TCC3`.
Q.E.D.

C.123.37 Memory_Change_2.unchanged_memory_invariant_unchanged_write_list

Terse proof for `unchanged_memory_invariant_unchanged_write_list`.
`unchanged_memory_invariant_unchanged_write_list`:

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], s: State, raddr, waddr: Address,
        bl: list[Byte]):
       unchanged_memory_invariant?(pm, states,
                                   (memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ me
                                   singleton(raddr))
        ∧
        transformers_ok?(states,
                         (memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ memory_write
         ∧
         transformers_ok?(states, singleton(expr_2_super(memory_read(pm)(raddr)))) ∧
          side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                        memory_write_side_effect(pm))
           ∧ states(s)
        ⊃
        data(memory_read(pm)(raddr)(state(memory_write_list(pm)(waddr, bl)(s)))) =
         data(memory_read(pm)(raddr)(s))
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of memory_write_list,
Using lemma expr_transformers_ok_ok,
Installing automatic rewrites from: union_right memory_write_side_effect_super_transformers_memory_write_side_effect subset_equal singleton has_next_state union_subset1
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ##,
Using lemma side_effect_content_unchanged_content[State],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -4,
Hiding formulas: -4,
Using lemma expr_unchanged_memory_invariant_next_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma unchanged_memory_invariant_unchanged_write_list_nse,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`unchanged_memory_invariant_unchanged_write_list.1:`

{-1} transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-2} states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-3} data(memory_read(pm′)
(raddr′)
(state(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))))
=
data(memory_read(pm′)
(raddr′)(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))))
{-4} unchanged_memory_invariant?(pm′, states′,
(memory_write_transformers(pm′, address_block(waddr′, lengt
singleton(raddr′))
{-5} states′(s′)
{-6} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-7} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-8} transformers_ok?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mer

---

{1} data(memory_read(pm′)
(raddr′)
(state(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))))
= data(memory_read(pm′)(raddr′)(s′))

Hiding formulas: -2,

Using lemma expr_unchanged_memory_invariant_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_invariant_unchanged_write_list.1.1:`

{-1} unchanged_memory_invariant?(pm′, states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ m
singleton(raddr′))
{-2} states′(s′)
{-3} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-4} transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-5} data(memory_read(pm′)
(raddr′)
(state(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))))
=
data(memory_read(pm′)
(raddr′)(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))))
{-6} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-7} transformers_ok?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write_

{1} OK?(memory_read(pm′)
(raddr′)(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))))
{2} data(memory_read(pm′)
(raddr′)
(state(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))))
= data(memory_read(pm′)(raddr′)(s′))

Hiding formulas: -1, -5, 2,

Using lemma expr_transformers_ok_ok[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_unchanged_write_list.1.1`.

`unchanged_memory_invariant_unchanged_write_list.1.2:`

```
{-1}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_write_transformers(pm′, address_block(waddr′, lengt
                                  singleton(raddr′))
{-2}  states′(s′)
{-3}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-4}  transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-5}  data(memory_read(pm′)
           (raddr′)
           (state(memory_write_list_nse(pm′)(waddr′, bl′)
                                            (state(memory_write_side_effect(pm′)
                                                   (waddr′, bl′, FALSE)(s′))))))
      =
      data(memory_read(pm′)
           (raddr′)(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))))
{-6}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-7}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
------------------------------------------------------------------------------------------------
{1}   OK?(memory_read(pm′)(raddr′)(s′))
{2}   data(memory_read(pm′)
           (raddr′)
           (state(memory_write_list_nse(pm′)(waddr′, bl′)
                                            (state(memory_write_side_effect(pm′)
                                                   (waddr′, bl′, FALSE)(s′))))))
      = data(memory_read(pm′)(raddr′)(s′))
```

Hiding formulas: -1, -5, 2,
Using lemma expr_transformers_ok_ok[State, Byte],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_write_list.1.2`.
`unchanged_memory_invariant_unchanged_write_list.2:`

```
{-1}  transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-2}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-3}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_write_transformers(pm′, address_block(waddr′, lengt
                                  singleton(raddr′))
{-4}  states′(s′)
{-5}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-6}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-7}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
------------------------------------------------------------------------------------------------
{1}   transformers_ok?(states′,
                       memory_write_transformers(pm′, address_block(waddr′, length(bl′))))
{2}   data(memory_read(pm′)
           (raddr′)
           (state(memory_write_list_nse(pm′)(waddr′, bl′)
                                            (state(memory_write_side_effect(pm′)
                                                   (waddr′, bl′, FALSE)(s′))))))
      = data(memory_read(pm′)(raddr′)(s′))
```

Keeping (-7 1) and hiding *,

Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_write_list.2`.
`unchanged_memory_invariant_unchanged_write_list.3`:

```
{-1}  transformers_ok?(states′, singleton(expr_2_super(memory_read(pm′)(raddr′))))
{-2}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-3}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ m
                                  singleton(raddr′))
{-4}  states′(s′)
{-5}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-6}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-7}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write_
------------------------------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm′, states′,
                                  memory_write_transformers(pm′,
                                                            ad-
      dress_block(waddr′, length(bl′))),
                                  singleton(raddr′))
{2}   data(memory_read(pm′)
           (raddr′)
           (state(memory_write_list_nse(pm′)(waddr′, bl′)
                                            (state(memory_write_side_effect(pm′)
                                                   (waddr′, bl′, FALSE)(s′))))))
        = data(memory_read(pm′)(raddr′)(s′))
```

Keeping (-3 1) and hiding *,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_invariant_unchanged_write_list.3`.
Q.E.D.

C.123.38 Memory_Change_2.unchanged_memory_invariant_read_list

Terse proof for `unchanged_memory_invariant_read_list`.
`unchanged_memory_invariant_read_list`:

```
------------------------------------------------------------------------------------------------------------
{1}   ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], addr: Ad-
      dress,
        size: nat):
        unchanged_memory_invariant?(pm, states,
                                    (memory_read_transformers(pm, address_block(addr, size)) ∪ memory_re
                                    addresses)
          ∧
          transformers_ok?(states,
                           (memory_read_transformers(pm, addresses) ∪ memory_read_side_effect_super_transfor
           ∧ (address_block(addr, size) ⊆ addresses)
          ⊃
          unchanged_memory_invariant?(pm, states,
                                      singleton(expr_2_super(memory_read_list(pm)(addr, size))),
                                      addresses)
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: result_pred state_expr_2_super ok_expr_2_super has_next_state memory_read_transformers_mono subset_bigger_union_left memory_read_side_effect_super_transformers_mono subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right union_left union_right transformers_ok_union_transformers memory_read_transformers_memory_read memory_read_side_effect_super_transformers_memory_read_side_effect

Using lemma transformers_ok_mono_transformers,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

`unchanged_memory_invariant_read_list.1:`

{-1}	transformers_ok?(states′, (memory_read_transformers(pm′, addresses′) ∪ memory_read_side_effect_supe
{-2}	transformers_ok?(states′, memory_read_side_effect_super_transformers(pm′, address_block(addr′, size′)))
{-3}	transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr′, size′)))
{-4}	size′ ≥ 0
{-5}	unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(addr′, size′)) ∪ addresses′)
{-6}	(address_block(addr′, size′) ⊆ addresses′)
{1}	transformer_invariant?(states′, singleton(expr_2_super(memory_read_list(pm′)(addr′, size′))))

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -2,

Hiding formulas: -2,

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_read_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_next_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_read_list.1`.

`unchanged_memory_invariant_read_list.2:`

{-1} transformers_ok?(states′, (memory_read_transformers(pm′, addresses′) ∪ memory_read_side_effect_super_transform
{-2} transformers_ok?(states′, memory_read_side_effect_super_transformers(pm′, address_block(addr′, size′)))
{-3} transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr′, size′)))
{-4} size′ ≥ 0
{-5} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(addr′, size′)) ∪ memory_re addresses′)
{-6} (address_block(addr′, size′) ⊆ addresses′)

{1} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
singleton(expr_2_super(memory_read_list(pm′)(addr′, size′)))(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -4,

Hiding formulas: -4,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_invariant_read_list.2.1:`

```
{-1}  transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr′, size′)))
{-2}  states′(s′)
{-3}  states′(state(memory_read_list_nse(pm′)(addr′, size′)(s′)))
{-4}  length(data(memory_read_list_nse(pm′)(addr′, size′)(s′))) = size′
{-5}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(addr′, size′))
                                  addresses′)
{-6}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′, ad-
      dress_block(addr′, size′)),
                                  addresses′)
{-7}  addresses′(a′)
{-8}  OK?(memory_read_list_nse(pm′)(addr′, size′)(s′))
{-9}  OK?(memory_read_side_effect(pm′)
            (addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)), FALSE)
            (state(memory_read_list_nse(pm′)(addr′, size′)(s′))))
{-10} OK?(memory_read(pm′)(a′)(s′))
{-11} OK?(memory_read(pm′)
            (a′)
            (state(memory_read_side_effect(pm′)
                        (addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
                          FALSE)
                        (state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
{-12} transformers_ok?(states′,
                       (memory_read_transformers(pm′, addresses′) ∪ memory_read_side_effect_sup
{-13} transformers_ok?(states′,
                       memory_read_side_effect_super_transformers(pm′,
                                                                  ad-
      dress_block(addr′, size′)))
{-14} size′ ≥ 0
{-15} (address_block(addr′, size′) ⊆ addresses′)
-------------------------------------------------------------------------------
 {1}  data(memory_read(pm′)
             (a′)
             (state(memory_read_side_effect(pm′)
                         (addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
                           FALSE)
                         (state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
      = data(memory_read(pm′)(a′)(s′))
```

Using lemma expr_unchanged_memory_invariant_unchanged[State, list[Byte]],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_invariant_read_list.2.1.1:

{-1} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(addr′, size′)) ∪ memory_r
addresses′)
{-2} states′(state(memory_read_list_nse(pm′)(addr′, size′)(s′)))
{-3} addresses′(a′)
{-4} OK?(memory_read_side_effect(pm′)
(addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)), FALSE)
(state(memory_read_list_nse(pm′)(addr′, size′)(s′))))
{-5} OK?(memory_read(pm′)
(a′)
(state(memory_read_side_effect(pm′)
(addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
FALSE)
(state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
{-6} data(memory_read(pm′)
(a′)
(state(memory_read_side_effect(pm′)
(addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
FALSE)
(state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
=
data(memory_read(pm′)
(a′)(state(memory_read_list_nse(pm′)(addr′, size′)(s′))))
{-7} transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr′, size′)))
{-8} states′(s′)
{-9} length(data(memory_read_list_nse(pm′)(addr′, size′)(s′))) = size′
{-10} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′, address_block(addr′, size′)),
addresses′)
{-11} OK?(memory_read_list_nse(pm′)(addr′, size′)(s′))
{-12} OK?(memory_read(pm′)(a′)(s′))
{-13} transformers_ok?(states′,
(memory_read_transformers(pm′, addresses′) ∪ memory_read_side_effect_super_transfor
{-14} transformers_ok?(states′,
memory_read_side_effect_super_transformers(pm′,
address_block(addr′, size′)))
{-15} size′ ≥ 0
{-16} (address_block(addr′, size′) ⊆ addresses′)

{1} data(memory_read(pm′)
(a′)
(state(memory_read_side_effect(pm′)
(addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
FALSE)
(state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
= data(memory_read(pm′)(a′)(s′))

Replacing using formula -6,
Hiding formulas: -6,
Using lemma unchanged_memory_invariant_unchanged_read_list_nse,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-13 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_read_list.2.1.1`.

`unchanged_memory_invariant_read_list.2.1.2`:

```
{-1}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(addr′, size′)) ∪
                                  addresses′)
{-2}  states′(state(memory_read_list_nse(pm′)(addr′, size′)(s′)))
{-3}  addresses′(a′)
{-4}  OK?(memory_read_side_effect(pm′)
                  (addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)), FALSE)
                  (state(memory_read_list_nse(pm′)(addr′, size′)(s′))))
{-5}  OK?(memory_read(pm′)
                  (a′)
                  (state(memory_read_side_effect(pm′)
                                    (addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
                                      FALSE)
                                    (state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
{-6}  transformers_ok?(states′, memory_read_transformers(pm′, address_block(addr′, size′)))
{-7}  states′(s′)
{-8}  length(data(memory_read_list_nse(pm′)(addr′, size′)(s′))) = size′
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′, ad-
      dress_block(addr′, size′)),
                                  addresses′)
{-10} OK?(memory_read_list_nse(pm′)(addr′, size′)(s′))
{-11} OK?(memory_read(pm′)(a′)(s′))
{-12} transformers_ok?(states′,
                       (memory_read_transformers(pm′, addresses′) ∪ memory_read_side_effect_sup
{-13} transformers_ok?(states′,
                       memory_read_side_effect_super_transformers(pm′,
                                                                  ad-
      dress_block(addr′, size′)))
{-14} size′ ≥ 0
{-15} (address_block(addr′, size′) ⊆ addresses′)
-------------------------------------------------------------------------------
 {1}  OK?(memory_read(pm′)(a′)(state(memory_read_list_nse(pm′)(addr′, size′)(s′))))
 {2}  data(memory_read(pm′)
                  (a′)
                  (state(memory_read_side_effect(pm′)
                                    (addr′, data(memory_read_list_nse(pm′)(addr′, size′)(s′)),
                                      FALSE)
                                    (state(memory_read_list_nse(pm′)(addr′, size′)(s′))))))
       = data(memory_read(pm′)(a′)(s′))
```

Keeping (-2 -3 -10 -12 1) and hiding *,

Using lemma expr_transformers_ok_ok[State, Byte],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_invariant_read_list.2.1.2`.

`unchanged_memory_invariant_read_list.2.2`:

```
{-1}  transformers_ok?(states', memory_read_transformers(pm', address_block(addr', size')))
{-2}  states'(s')
{-3}  unchanged_memory_invariant?(pm', states',
                                  (memory_read_transformers(pm', address_block(addr', size')) ∪ memory_r
                                  addresses')
{-4}  unchanged_memory_invariant?(pm', states',
                                  memory_read_transformers(pm', ad-
      dress_block(addr', size')),
                                  addresses')
{-5}  addresses'(a')
{-6}  OK?(memory_read_list_nse(pm')(addr', size')(s'))
{-7}  OK?(memory_read_side_effect(pm')
                (addr', data(memory_read_list_nse(pm')(addr', size')(s')), FALSE)
                (state(memory_read_list_nse(pm')(addr', size')(s'))))
{-8}  OK?(memory_read(pm')(a')(s'))
{-9}  OK?(memory_read(pm')
                (a')
                (state(memory_read_side_effect(pm')
                              (addr', data(memory_read_list_nse(pm')(addr', size')(s')),
                                FALSE)
                              (state(memory_read_list_nse(pm')(addr', size')(s'))))))
{-10} transformers_ok?(states',
                       (memory_read_transformers(pm', addresses') ∪ memory_read_side_effect_super_transfor
{-11} transformers_ok?(states',
                       memory_read_side_effect_super_transformers(pm',
                                                                  ad-
      dress_block(addr', size')))
{-12} size' ≥ 0
{-13} (address_block(addr', size') ⊆ addresses')
-------------------------------------------------------------------------------
 {1}  transformer_invariant?(states',
                             memory_read_transformers(pm', ad-
      dress_block(addr', size')))
 {2}  data(memory_read(pm')
           (a')
           (state(memory_read_side_effect(pm')
                         (addr', data(memory_read_list_nse(pm')(addr', size')(s')),
                           FALSE)
                         (state(memory_read_list_nse(pm')(addr', size')(s'))))))
        = data(memory_read(pm')(a')(s'))
```

Keeping (-4 1) and hiding *,

Forward chaining on unchanged_memory_invariant_invariant,

This completes the proof of `unchanged_memory_invariant_read_list.2.2`.

Q.E.D.

C.123.39 Memory_Change_2.unchanged_memory_read_list_write_list_nse_TCC1

Terse proof for `unchanged_memory_read_list_write_list_nse_TCC1`.

unchanged_memory_read_list_write_list_nse_TCC1:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
s: State):
unchanged_memory_invariant?(pm, states,
memory_read_transformers(pm,
address_block(waddr, length(bl))),
address_block(waddr, length(bl)))
∧
unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
transformers_ok?(states,
memory_read_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧ states(s)
⊃
OK?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s)) ∨
Exception?[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s))

Repeatedly Skolemizing and flattening,

Using lemma memory_write_list_ok_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using unchanged_memory_write_invariant_transformer_invariant, matching in *,

This completes the proof of unchanged_memory_read_list_write_list_nse_TCC1.

Q.E.D.

C.123.40 Memory_Change_2.unchanged_memory_read_list_write_list_nse_TCC2

Terse proof for unchanged_memory_read_list_write_list_nse_TCC2.

`unchanged_memory_read_list_write_list_nse_TCC2:`

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
        s: State):
       unchanged_memory_invariant?(pm, states,
                                   memory_read_transformers(pm,
                                                            address_block(waddr, length(bl))),
                                   address_block(waddr, length(bl)))
         ∧
         unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
          changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
           transformers_ok?(states,
                            memory_read_transformers(pm, address_block(waddr, length(bl))))
             ∧
             transformers_ok?(states,
                              memory_write_transformers(pm, address_block(waddr, length(bl))))
               ∧ states(s)
         ⊃
         OK?[State, list[Byte]]
           (memory_read_list_nse[State]
              (pm)(waddr, length[Byte](bl))
              (state[State, Unit](memory_write_list_nse[State](pm)(waddr, bl)(s))))
```

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_nse_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`unchanged_memory_read_list_write_list_nse_TCC2.1:`

```
{-1}  transformers_ok?(states',
                       memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-2}  unchanged_memory_invariant?(pm', states',
                                  memory_read_transformers(pm',
                                                           address_block(waddr', length(bl'))),
                                  address_block(waddr', length(bl')))
{-3}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}  transformers_ok?(states',
                       memory_write_transformers(pm', address_block(waddr', length(bl'))))
{-6}  states'(s')
------------------------------------------------------------------------------------------
 {1}  states'(state[State, Unit](memory_write_list_nse[State](pm')(waddr', bl')(s')))
 {2}  OK?[State, list[Byte]]
        (memory_read_list_nse[State]
           (pm')(waddr', length[Byte](bl'))
           (state[State, Unit]
              (memory_write_list_nse[State](pm')(waddr', bl')(s'))))
```

Hiding formulas: 2,

Using lemma unchanged_memory_write_invariant_transformer_invariant,

Simplifying, rewriting, and recording with decision procedures,

Using lemma transformer_invariant_write_list_nse,

Simplifying, rewriting, and recording with decision procedures,

Using lemma memory_write_list_ok_nse,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: has_next_state

Rewriting using expr_result_pred_next_state, matching in *,

This completes the proof of `unchanged_memory_read_list_write_list_nse_TCC2.1`.

`unchanged_memory_read_list_write_list_nse_TCC2.2`:

```
{-1}  transformers_ok?(states',
                       memory_read_transformers(pm', address_block(waddr', length(bl'))))
{-2}  unchanged_memory_invariant?(pm', states',
                                  memory_read_transformers(pm',
                                                           address_block(waddr', length(bl'))),
                                  address_block(waddr', length(bl')))
{-3}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-4}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-5}  transformers_ok?(states',
                       memory_write_transformers(pm', address_block(waddr', length(bl'))))
{-6}  states'(s')
|-----------------------------------------------------------------------------------------
{1}   transformer_invariant?(states',
                             memory_read_transformers(pm',
                                                      address_block(waddr',
                                                                    length[Byte](bl'))))
{2}   OK?[State, list[Byte]]
          (memory_read_list_nse[State]
                (pm')(waddr', length[Byte](bl'))
                (state[State, Unit]
                      (memory_write_list_nse[State](pm')(waddr', bl')(s'))))
```

Forward chaining on unchanged_memory_invariant_invariant,

This completes the proof of `unchanged_memory_read_list_write_list_nse_TCC2.2`.

Q.E.D.

C.123.41 Memory_Change_2.unchanged_memory_read_list_write_list_nse

Terse proof for `unchanged_memory_read_list_write_list_nse`.

`unchanged_memory_read_list_write_list_nse:`

{1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
s: State):
unchanged_memory_invariant?(pm, states,
memory_read_transformers(pm,
address_block(waddr, length(bl))),
address_block(waddr, length(bl)))
∧
unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
transformers_ok?(states,
memory_read_transformers(pm, address_block(waddr, length(bl))))
∧
transformers_ok?(states,
memory_write_transformers(pm, address_block(waddr, length(bl))))
∧ states(s)
⊃
data(memory_read_list_nse(pm)(waddr, length(bl))
(state(memory_write_list_nse(pm)(waddr, bl)(s))))
= bl

For the top quantifier in 1, we introduce Skolem constants: (pm′ states′ _ _ _),
Inducting on bl on formula 1,
we get 4 subgoals:
`unchanged_memory_read_list_write_list_nse.1:`

{1} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr, length(null))),
address_block(waddr, length(null)))
∧
unchanged_memory_write_invariant?(pm′, states′, address_block(waddr, length(null))) ∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(null))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′, address_block(waddr, length(null))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′, address_block(waddr, length(null))))
∧ states′(s)
⊃
data(memory_read_list_nse(pm′)(waddr, length(null))
(state(memory_write_list_nse(pm′)(waddr, null)(s))))
= null

Repeatedly Skolemizing and flattening,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.1`.

unchanged_memory_read_list_write_list_nse.2:

{1} ∀ (cons1_var: Byte, cons2_var: list[Byte]):
(∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block
(waddr, length(cons2_var))),
address_block(waddr, length(cons2_var)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons2_var))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var))))
∧ states′(s)
⊃
data(memory_read_list_nse(pm′)(waddr, length(cons2_var))
(state(memory_write_list_nse(pm′)
(waddr, cons2_var)
(s))))
= cons2_var)
⊃
(∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block
(waddr,
length
(cons
(cons1_var, cons2_var)))),
address_block(waddr,
length(cons(cons1_var, cons2_var))))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr,
length(cons
(cons1_var, cons2_var))))
∧
changed_memory_invariant?(pm′, states′,
address_block(waddr,
length(cons(cons1_var, cons2_var))))
∧
transformers_ok?(states′,
memory_read_transformers(pm′,
address_block(waddr,
length
(cons
(cons1_var,
cons2_var)))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length
(cons

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: address_block address_block_subset_1 ##! length subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right memory_read_transformers_memory_read memory_read_transformers_mono memory_write_transformers_memory_write memory_write_transformers_mono transformer_invariant_mono_transformers expr_transformer_invariant_next_ok ok_result has_next_state state_expr_2_super ok_expr_2_super

Using lemma unchanged_memory_write_invariant_transformer_invariant,

Simplifying, rewriting, and recording with decision procedures,

Using lemma memory_write_list_ok_nse,

Simplifying, rewriting, and recording with decision procedures,

Using lemma unchanged_memory_read_list_write_list_nse_TCC2,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_write_list_nse,

Expanding the definition of memory_read_list_nse,

Expanding the definition of memory_write_list_nse,

Expanding the definition of memory_write_list_nse,

Expanding the definition of memory_read_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Applying extensionality,

we get 2 subgoals:

unchanged_memory_read_list_write_list_nse.2.1:

{-1} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-2} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-3} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write(pm′)
(waddr′, cons1_var′)
(s′))))))))

{-4} OK?[State, list[Byte]]
(OK(state(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_wr
(pm′)
(waddr′,
cons1_v
(s′))))))))
cons(data(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)
(s′))))))),
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse
(pm′)
(waddr′ + 1, cons2_var′)
(state
(memory_write(pm′)
(waddr′, cons1_var′)
(s′)))))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))))

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons2_var′))),
address_block(waddr, length(cons2_var′)))
∧

Hiding formulas: -2, -3, -4, -7,
Using lemma expr_transformer_invariant_next_ok,
Simplifying, rewriting, and recording with decision procedures,
Using lemma expr_transformers_ok_ok,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_unchanged_write_list_nse,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:

`unchanged_memory_read_list_write_list_nse.2.1.1:`

{-1} $\mathrm{states}'(\mathrm{state}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s')))$
{-2} $\mathrm{data}(\mathrm{memory\_read}(\mathrm{pm}')(\mathrm{waddr}')(\mathrm{state}(\mathrm{memory\_write\_list\_nse}(\mathrm{pm}')(\mathrm{waddr}'+1, \mathrm{cons2\_var}')(\mathrm{state}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s'))))))$
$=$
$\mathrm{data}(\mathrm{memory\_read}(\mathrm{pm}')(\mathrm{waddr}')(\mathrm{state}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s'))))$
{-3} $\mathrm{OK?}(\mathrm{memory\_read}(\mathrm{pm}')(\mathrm{waddr}')(\mathrm{state}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s'))))$
{-4} $\mathrm{OK?}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s'))$
{-5} $\mathrm{OK?}(\mathrm{memory\_write\_list\_nse}(\mathrm{pm}')(\mathrm{waddr}'+1, \mathrm{cons2\_var}')(\mathrm{state}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s'))))$
{-6} $\mathrm{transformer\_invariant?}(\mathrm{states}', \mathrm{memory\_write\_transformers}(\mathrm{pm}', \mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{cons2\_v$
{-7} $\mathrm{unchanged\_memory\_invariant?}(\mathrm{pm}', \mathrm{states}', \mathrm{memory\_read\_transformers}(\mathrm{pm}', \mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{co$
$\mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{cons2\_var}')))$
{-8} $\mathrm{unchanged\_memory\_write\_invariant?}(\mathrm{pm}', \mathrm{states}', \mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{cons2\_var}')))$
{-9} $\mathrm{changed\_memory\_invariant?}(\mathrm{pm}', \mathrm{states}', \mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{cons2\_var}')))$
{-10} $\mathrm{transformers\_ok?}(\mathrm{states}', \mathrm{memory\_read\_transformers}(\mathrm{pm}', \mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{cons2\_var}'))))$
{-11} $\mathrm{transformers\_ok?}(\mathrm{states}', \mathrm{memory\_write\_transformers}(\mathrm{pm}', \mathrm{address\_block}(\mathrm{waddr}', 1+\mathrm{length}(\mathrm{cons2\_var}'))))$
{-12} $\mathrm{states}'(s')$

{1} $\mathrm{data}(\mathrm{memory\_read}(\mathrm{pm}')(\mathrm{waddr}')(\mathrm{state}[\mathrm{State}, \mathrm{Unit}](\mathrm{memory\_write\_list\_nse}(\mathrm{pm}')(\mathrm{waddr}'+1, \mathrm{cons2\_var}')(\mathrm{state}(\mathrm{memory\_write}(\mathrm{pm}')(\mathrm{waddr}', \mathrm{cons1\_var}')(s'))))))$
$= \mathrm{cons1\_var}'$

Replacing using formula -2,

Keeping (-9 1) and hiding *,

Expanding the definition of changed_memory_invariant?,

Applying disjunctive simplification to flatten sequent,

Rewriting using -2, matching in *,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.1.1`.

`unchanged_memory_read_list_write_list_nse.2.1.2`:

```
{-1}  states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-2}  OK?(memory_read(pm')
            (waddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
                                  (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5}  transformer_invariant?(states',
                               memory_write_transformers(pm',
                                                         address_block(waddr',
                                                                       1 + length(cons2_var'))))
{-6}  unchanged_memory_invariant?(pm', states',
                                  memory_read_transformers(pm',
                                                           address_block(waddr',
                                                                         1 + length(cons2_var'))),
                                  address_block(waddr', 1 + length(cons2_var')))
{-7}  unchanged_memory_write_invariant?(pm', states',
                                        address_block(waddr', 1 + length(cons2_var')))
{-8}  changed_memory_invariant?(pm', states',
                                address_block(waddr', 1 + length(cons2_var')))
{-9}  transformers_ok?(states',
                       memory_read_transformers(pm',
                                                address_block(waddr',
                                                              1 + length(cons2_var'))))
{-10} transformers_ok?(states',
                       memory_write_transformers(pm',
                                                 address_block(waddr',
                                                               1 + length(cons2_var'))))
{-11} states'(s')
-------------------------------------------------------------------------------
 {1}  transformers_ok?(states', singleton(expr_2_super(memory_read(pm')(waddr'))))
 {2}  data(memory_read(pm')
           (waddr')
           (state[State, Unit]
                  (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
                                         (state(memory_write(pm')
                                                            (waddr', cons1_var')(s'))))))
        = cons1_var'
```

Keeping (-9 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.1.2`.

`unchanged_memory_read_list_write_list_nse.2.1.3:`

```
{-1}  states′(state(memory_write(pm′)(waddr′, cons1_var′)(s′)))
{-2}  OK?(memory_read(pm′)
            (waddr′)(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-3}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-4}  OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                  (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-5}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_v
{-6}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(waddr′,
                                                                         1 + length(con
                                  address_block(waddr′, 1 + length(cons2_var′)))
{-7}  unchanged_memory_write_invariant?(pm′, states′,
                                        address_block(waddr′, 1 + length(cons2_var′)))
{-8}  changed_memory_invariant?(pm′, states′,
                                address_block(waddr′, 1 + length(cons2_var′)))
{-9}  transformers_ok?(states′,
                       memory_read_transformers(pm′,
                                                address_block(waddr′,
                                                              1 + length(cons2_var′))))
{-10} transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-11} states′(s′)
-------------------------------------------------------------------------------
 {1}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′ + 1,
                                                               length(cons2_var′))))
 {2}  data(memory_read(pm′)
             (waddr′)
             (state[State, Unit]
                    (memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                               (state(memory_write(pm′)
                                                                  (waddr′, cons1_var′)(s′))))))
      = cons1_var′
```

Keeping (-10 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.1.3`.

`unchanged_memory_read_list_write_list_nse.2.1.4:`

```
{-1}  states'(state(memory_write(pm')(waddr', cons1_var')(s')))
{-2}  OK?(memory_read(pm')
            (waddr')(state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-4}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
                                  (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-5}  transformer_invariant?(states',
                               memory_write_transformers(pm',
                                                         address_block(waddr',
                                                                       1 + length(cons2_var'))))
{-6}  unchanged_memory_invariant?(pm', states',
                                  memory_read_transformers(pm',
                                                           address_block(waddr',
                                                                         1 + length(cons2_var'))),
                                  address_block(waddr', 1 + length(cons2_var')))
{-7}  unchanged_memory_write_invariant?(pm', states',
                                        address_block(waddr', 1 + length(cons2_var')))
{-8}  changed_memory_invariant?(pm', states',
                                address_block(waddr', 1 + length(cons2_var')))
{-9}  transformers_ok?(states',
                       memory_read_transformers(pm',
                                                address_block(waddr',
                                                              1 + length(cons2_var'))))
{-10} transformers_ok?(states',
                       memory_write_transformers(pm',
                                                 address_block(waddr',
                                                               1 + length(cons2_var'))))
{-11} states'(s')
-------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm', states',
                                  memory_write_transformers(pm',
                                                            ad-
      dress_block(waddr' + 1,
                                                                length(cons2_var'))),
                                  singleton(waddr'))
 {2}  data(memory_read(pm')
           (waddr')
           (state[State, Unit]
                  (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
                                             (state(memory_write(pm')
                                                   (waddr', cons1_var')(s'))))))
        = cons1_var'
```

Hiding formulas: 2,

Using lemma unchanged_memory_write_invariant_unchanged_memory_invariant,

Simplifying, rewriting, and recording with decision procedures,

Keeping 1 and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.1.4`.

unchanged_memory_read_list_write_list_nse.2.2:

{-1} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-2} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-3} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-4} OK?[State, list[Byte]]
(OK(state(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(wa
c
(sta
(

cons(data(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)
(s′))))))),
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse
(pm′)
(waddr′ + 1, cons2_va
(state
(memory_write(pm′)
(waddr′, cons1_v
(s′))))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_va

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧

Hiding formulas: -4,

Using lemma unchanged_memory_read_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

```
unchanged_memory_read_list_write_list_nse.2.2.1:

{-1}  data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
                                    (state(memory_read(pm′)
                                                      (waddr′)
                                                      (state[State, Unit]
                                                            (memory_write_list_nse(pm′)
                                                                                  (waddr′ + 1,
                                                                                    cons2_var′)
                                                                                  (state
                                                                                    (memory_writ
                                                                                      (waddr′, co
                                                                                      (s′))))))))
      =
      data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
                                    (state[State, Unit]
                                          (memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                                                    (state(memory_write(pm′)
                                                                                       (waddr′, cons1_var′
                                                                                       (s′))))))
{-2}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-3}  OK?(memory_read(pm′)
                     (waddr′)
                     (state[State, Unit]
                           (memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                                 (state(memory_write(pm′)
                                                                    (waddr′, cons1_var′)(s′))))))
{-4}  OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
                                   (state(memory_read(pm′)
                                                     (waddr′)
                                                     (state[State, Unit]
                                                           (memory_write_list_nse(pm′)
                                                                                 (waddr′ + 1,
                                                                                   cons2_var′)
                                                                                 (state
                                                                                   (memory_write
                                                                                     (waddr′, cor
                                                                                     (s′))))))))
{-5}  OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                    (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-6}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_va
{-7}  ∀ (waddr: Address, s: State):
        unchanged_memory_invariant?(pm′, states′,
                                    memory_read_transformers(pm′,
                                                             address_block(waddr,
                                                                           length(cons
                                    address_block(waddr, length(cons2_var′)))
          ∧
          unchanged_memory_write_invariant?(pm′, states′,
                                            address_block(waddr, length(cons2_var′)))
           ∧
           changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
            transformers_ok?(states′,
                             memory_read_transformers(pm′,
                                                      ad-
      dress_block(waddr, length(cons2_var′))))
             ∧
             transformers_ok?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr,
                                                                     length(cons2_var′))
              ∧ states′(s)
```

Replacing using formula -1,

Rewriting using -7, matching in *,

we get 6 subgoals:

unchanged_memory_read_list_write_list_nse.2.2.1.1:

{-1} data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_writ
(waddr′, c
(s′))))))))
=
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′
(s′))))))

{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-3} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-4} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)

Keeping (-8 1) and hiding *,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.1`.

unchanged_memory_read_list_write_list_nse.2.2.1.2:

{-1} data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_writ
(waddr′, c
(s′))))))))
=
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′
(s′))))))

{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-3} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-4} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)

Keeping (-9 1) and hiding *,

Using lemma unchanged_memory_write_invariant_mono_addresses,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.2`.

unchanged_memory_read_list_write_list_nse.2.2.1.3:

{-1} data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_writ
(waddr′, c
(s′))))))))
=
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′
(s′))))))

{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-3} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-4} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)

Keeping (-10 1) and hiding *,

Using lemma changed_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.3`.

unchanged_memory_read_list_write_list_nse.2.2.1.4:

{-1} data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_writ
(waddr′, c
(s′))))))))
=
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′
(s′))))))

{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-3} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-4} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)

Keeping (-11 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.4`.

```
unchanged_memory_read_list_write_list_nse.2.2.1.5:
```

{-1} data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_writ
(waddr′, c
(s′))))))))
=
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′
(s′))))))

{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-3} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-4} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)

Keeping (-12 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.5`.

unchanged_memory_read_list_write_list_nse.2.2.1.6:

{-1} data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_writ
(waddr′, c
(s′))))))))
=
data(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′
(s′))))))

{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))

{-3} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))

{-4} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))

{-5} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))

{-6} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v

{-7} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)

Using lemma expr_transformer_invariant_next_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.1.6`.

```
unchanged_memory_read_list_write_list_nse.2.2.2:

{-1}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2}  OK?(memory_read(pm′)
                (waddr′)
                (state[State, Unit]
                      (memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                                 (state(memory_write(pm′)
                                                                    (waddr′, cons1_var′)(s′))))))
{-3}  OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
                                   (state(memory_read(pm′)
                                                     (waddr′)
                                                     (state[State, Unit]
                                                           (memory_write_list_nse(pm′)
                                                                                 (waddr′ + 1,
                                                                                   cons2_var′)
                                                                                 (state
                                                                                   (memory_write
                                                                                     (waddr′, co
                                                                                     (s′)))))))))
{-4}  OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                    (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-5}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_v
{-6}  ∀ (waddr: Address, s: State):
        unchanged_memory_invariant?(pm′, states′,
                                    memory_read_transformers(pm′,
                                                             address_block(waddr,
                                                                           length(cons
                                    address_block(waddr, length(cons2_var′)))
          ∧
          unchanged_memory_write_invariant?(pm′, states′,
                                            address_block(waddr, length(cons2_var′)))
           ∧
           changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
            transformers_ok?(states′,
                             memory_read_transformers(pm′,
                                                      ad-
dress_block(waddr, length(cons2_var′))))
             ∧
             transformers_ok?(states′,
                              memory_write_transformers(pm′,
                                                        address_block(waddr,
                                                                      length(cons2_var′))
              ∧ states′(s)
          ⊃
          data(memory_read_list_nse(pm′)(waddr, length(cons2_var′))
                                        (state(memory_write_list_nse(pm′)(waddr, cons2_var′)
                                                                         (s))))
           = cons2_var′
{-7}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(waddr′,
                                                                         1 + length(co
                                  address_block(waddr′, 1 + length(cons2_var′)))
{-8}  unchanged_memory_write_invariant?(pm′, states′,
                                        address_block(waddr′, 1 + length(cons2_var′)))
{-9}  changed_memory_invariant?(pm′, states′,
                                address_block(waddr′, 1 + length(cons2_var′)))
{-10} transformers_ok?(states′,
                       memory_read_transformers(pm′,
                                                address_block(waddr′,
                                                              1 + length(cons2_var′))))
{-11} transformers_ok?(states′,
```

Hiding formulas: -2, -3, -6, 2,

Using lemma transformer_invariant_write_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`unchanged_memory_read_list_write_list_nse.2.2.2.1:`

{-1} result_pred(states′)
(expr_2_super(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′))
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-2} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-3} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-4} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))))
{-5} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))),
address_block(waddr′, 1 + length(cons2_var′)))
{-6} unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr′, 1 + length(cons2_var′)))
{-7} changed_memory_invariant?(pm′, states′,
address_block(waddr′, 1 + length(cons2_var′)))
{-8} transformers_ok?(states′,
memory_read_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))))
{-9} transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))))
{-10} states′(s′)

{1} states′(state⌊State, Unit⌋
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′)))))

Rewriting using expr_result_pred_next_state, matching in *,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.2.1`.

`unchanged_memory_read_list_write_list_nse.2.2.2.2:`

```
{-1}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2}  OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                  (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-3}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_va
{-4}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(waddr′,
                                                                         1 + length(cons
                                  address_block(waddr′, 1 + length(cons2_var′)))
{-5}  unchanged_memory_write_invariant?(pm′, states′,
                                        address_block(waddr′, 1 + length(cons2_var′)))
{-6}  changed_memory_invariant?(pm′, states′,
                                address_block(waddr′, 1 + length(cons2_var′)))
{-7}  transformers_ok?(states′,
                       memory_read_transformers(pm′,
                                                address_block(waddr′,
                                                              1 + length(cons2_var′))))
{-8}  transformers_ok?(states′,
                       memory_write_transformers(pm′,
                                                 address_block(waddr′,
                                                               1 + length(cons2_var′))))
{-9}  states′(s′)
------------------------------------------------------------------------------
{1}   states′(state(memory_write(pm′)(waddr′, cons1_var′)(s′)))
{2}   states′(state[State, Unit]
                   (memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                              (state(memory_write(pm′)
                                                                 (waddr′, cons1_var′)(s′)))))
```

Keeping (-3 1) and hiding *,

Using lemma expr_transformer_invariant_next_ok[State, Unit],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.2.2`.

`unchanged_memory_read_list_write_list_nse.2.2.2.3:`

```
{-1}  OK?(memory_write(pm')(waddr', cons1_var')(s'))
{-2}  OK?(memory_write_list_nse(pm')(waddr' + 1, cons2_var')
                                    (state(memory_write(pm')(waddr', cons1_var')(s'))))
{-3}  transformer_invariant?(states',
                                 memory_write_transformers(pm',
                                                           address_block(waddr',
                                                                         1 + length(cons2_var'))))
{-4}  unchanged_memory_invariant?(pm', states',
                                      memory_read_transformers(pm',
                                                               address_block(waddr',
                                                                             1 + length(cons2_var'))),
                                      address_block(waddr', 1 + length(cons2_var')))
{-5}  unchanged_memory_write_invariant?(pm', states',
                                            address_block(waddr', 1 + length(cons2_var')))
{-6}  changed_memory_invariant?(pm', states',
                                    address_block(waddr', 1 + length(cons2_var')))
{-7}  transformers_ok?(states',
                           memory_read_transformers(pm',
                                                    address_block(waddr',
                                                                  1 + length(cons2_var'))))
{-8}  transformers_ok?(states',
                           memory_write_transformers(pm',
                                                     address_block(waddr',
                                                                   1 + length(cons2_var'))))
{-9}  states'(s')
|-------------------------------------------------------------------------------
{1}   transformer_invariant?(states',
                                 memory_write_transformers(pm',
                                                           address_block(waddr' + 1,
                                                                         length(cons2_var'))))
{2}   states'(state[State, Unit]
                   (memory_write_list_nse(pm')(waddr' + 1, cons2_var')
                                              (state(memory_write(pm')
                                                    (waddr', cons1_var')(s')))))
```

Keeping (-3 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.2.3`.

```
unchanged_memory_read_list_write_list_nse.2.2.3:

{-1}  OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2}  OK?(memory_read(pm′)
                (waddr′)
                (state[State, Unit]
                      (memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                                 (state(memory_write(pm′)
                                                                    (waddr′, cons1_var′)(s′))))))
{-3}  OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
                                   (state(memory_read(pm′)
                                                     (waddr′)
                                                     (state[State, Unit]
                                                           (memory_write_list_nse(pm′)
                                                                                 (waddr′ + 1,
                                                                                   cons2_var′)
                                                                                 (state
                                                                                   (memory_write
                                                                                     (waddr′, co
                                                                                     (s′)))))))))
{-4}  OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
                                    (state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-5}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       address_block(waddr′,
                                                                     1 + length(cons2_v
{-6}  ∀ (waddr: Address, s: State):
        unchanged_memory_invariant?(pm′, states′,
                                    memory_read_transformers(pm′,
                                                             address_block(waddr,
                                                                           length(cons
                                    address_block(waddr, length(cons2_var′)))
          ∧
          unchanged_memory_write_invariant?(pm′, states′,
                                            address_block(waddr, length(cons2_var′)))
           ∧
           changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
            transformers_ok?(states′,
                             memory_read_transformers(pm′,
                                                      ad-
dress_block(waddr, length(cons2_var′))))
              ∧
              transformers_ok?(states′,
                               memory_write_transformers(pm′,
                                                         address_block(waddr,
                                                                       length(cons2_var′))
               ∧ states′(s)
          ⊃
          data(memory_read_list_nse(pm′)(waddr, length(cons2_var′))
                                        (state(memory_write_list_nse(pm′)(waddr, cons2_var′)
                                                                         (s))))
           = cons2_var′
{-7}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           address_block(waddr′,
                                                                         1 + length(co
                                  address_block(waddr′, 1 + length(cons2_var′)))
{-8}  unchanged_memory_write_invariant?(pm′, states′,
                                        address_block(waddr′, 1 + length(cons2_var′)))
{-9}  changed_memory_invariant?(pm′, states′,
                                address_block(waddr′, 1 + length(cons2_var′)))
{-10} transformers_ok?(states′,
                       memory_read_transformers(pm′,
                                                address_block(waddr′,
                                                              1 + length(cons2_var′))))
{-11} transformers_ok?(states′,
```

Keeping (-10 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.3`.

unchanged_memory_read_list_write_list_nse.2.2.4:

{-1} OK?(memory_write(pm′)(waddr′, cons1_var′)(s′))
{-2} OK?(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)
(waddr′, cons1_var′)(s′))))))
{-3} OK?(memory_read_list_nse(pm′)(waddr′ + 1, length(cons2_var′))
(state(memory_read(pm′)
(waddr′)
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′ + 1,
cons2_var′)
(state
(memory_write
(waddr′, co
(s′))))))))
{-4} OK?(memory_write_list_nse(pm′)(waddr′ + 1, cons2_var′)
(state(memory_write(pm′)(waddr′, cons1_var′)(s′))))
{-5} transformer_invariant?(states′,
memory_write_transformers(pm′,
address_block(waddr′,
1 + length(cons2_v
{-6} ∀ (waddr: Address, s: State):
unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr,
length(cons
address_block(waddr, length(cons2_var′)))
∧
unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr, length(cons2_var′)))
∧
changed_memory_invariant?(pm′, states′, address_block(waddr, length(cons2_var′))) ∧
transformers_ok?(states′,
memory_read_transformers(pm′,
ad-
dress_block(waddr, length(cons2_var′))))
∧
transformers_ok?(states′,
memory_write_transformers(pm′,
address_block(waddr,
length(cons2_var′))
∧ states′(s)
⊃
data(memory_read_list_nse(pm′)(waddr, length(cons2_var′))
(state(memory_write_list_nse(pm′)(waddr, cons2_var′)
(s))))
= cons2_var′
{-7} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr′,
1 + length(co
address_block(waddr′, 1 + length(cons2_var′)))
{-8} unchanged_memory_write_invariant?(pm′, states′,
address_block(waddr′, 1 + length(cons2_var′)))
{-9} changed_memory_invariant?(pm′, states′,
address_block(waddr′, 1 + length(cons2_var′)))
{-10} transformers_ok?(states′,
memory_read_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))))
{-11} transformers_ok?(states′,

Keeping (-7 1) and hiding *,

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

we get 2 subgoals:

`unchanged_memory_read_list_write_list_nse.2.2.4.1:`

{-1} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))),
address_block(waddr′, 1 + length(cons2_var′)))

{1} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr′ + 1,
length(cons2_var′))),
address_block(waddr′, 1 + length(cons2_var′)))

{2} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(waddr′ + 1, length(cons2_
address_block(waddr′, 1 + length(cons2_var′)))

Hiding formulas: 2,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.4.1`.

`unchanged_memory_read_list_write_list_nse.2.2.4.2:`

{-1} unchanged_memory_invariant?(pm′, states′,
memory_read_transformers(pm′,
address_block(waddr′,
1 + length(cons2_var′))),
address_block(waddr′, 1 + length(cons2_var′)))

{1} unchanged_memory_invariant?(pm′, states′,
singleton(expr_2_super(memory_read(pm′)(waddr′))),
address_block(waddr′, 1 + length(cons2_var′)))

{2} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(waddr′ + 1, length(cons2_
address_block(waddr′, 1 + length(cons2_var′)))

Hiding formulas: 2,

Using lemma unchanged_memory_invariant_mono,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list_nse.2.2.4.2`.

`unchanged_memory_read_list_write_list_nse.3:`

```
{1}  ∀ (bl: list[Byte], waddr: Address, s: State):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′,
                                                            address_block(waddr, length(bl))),
                                   address_block(waddr, length(bl)))
         ∧
         unchanged_memory_write_invariant?(pm′, states′, address_block(waddr, length(bl))) ∧
          changed_memory_invariant?(pm′, states′, address_block(waddr, length(bl))) ∧
           transformers_ok?(states′,
                            memory_read_transformers(pm′, address_block(waddr, length(bl))))
             ∧
             transformers_ok?(states′,
                              memory_write_transformers(pm′, address_block(waddr, length(bl))))
               ∧ states′(s)
         ⊃
         OK?[State, list[Byte]]
             (memory_read_list_nse[State]
                 (pm′)(waddr, length[Byte](bl))
                 (state[State, Unit](memory_write_list_nse[State](pm′)(waddr, bl)(s))))
{2}  ∀ (waddr: Address, s: State):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′,
                                                            address_block(waddr, length(bl′))),
                                   address_block(waddr, length(bl′)))
         ∧
         unchanged_memory_write_invariant?(pm′, states′, address_block(waddr, length(bl′))) ∧
          changed_memory_invariant?(pm′, states′, address_block(waddr, length(bl′))) ∧
           transformers_ok?(states′,
                            memory_read_transformers(pm′, address_block(waddr, length(bl′))))
             ∧
             transformers_ok?(states′,
                              memory_write_transformers(pm′, address_block(waddr, length(bl′))))
               ∧ states′(s)
         ⊃
         data(memory_read_list_nse(pm′)(waddr, length(bl′))
                                   (state(memory_write_list_nse(pm′)(waddr, bl′)(s))))
          = bl′
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_read_list_write_list_nse_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.3`.

`unchanged_memory_read_list_write_list_nse.4:`

```
{1}  ∀ (bl: list[Byte], waddr: Address, s: State):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′,
                                                            address_block(waddr, length(bl))),
                                   address_block(waddr, length(bl)))
         ∧
         unchanged_memory_write_invariant?(pm′, states′, address_block(waddr, length(bl))) ∧
          changed_memory_invariant?(pm′, states′, address_block(waddr, length(bl))) ∧
           transformers_ok?(states′,
                            memory_read_transformers(pm′, address_block(waddr, length(bl))))
             ∧
             transformers_ok?(states′,
                              memory_write_transformers(pm′, address_block(waddr, length(bl))))
               ∧ states′(s)
         ⊃
         OK?[State, Unit](memory_write_list_nse[State](pm′)(waddr, bl)(s)) ∨
          Exception?[State, Unit](memory_write_list_nse[State](pm′)(waddr, bl)(s))
{2}  ∀ (waddr: Address, s: State):
       unchanged_memory_invariant?(pm′, states′,
                                   memory_read_transformers(pm′,
                                                            address_block(waddr, length(bl′))),
                                   address_block(waddr, length(bl′)))
         ∧
         unchanged_memory_write_invariant?(pm′, states′, address_block(waddr, length(bl′))) ∧
          changed_memory_invariant?(pm′, states′, address_block(waddr, length(bl′))) ∧
           transformers_ok?(states′,
                            memory_read_transformers(pm′, address_block(waddr, length(bl′))))
             ∧
             transformers_ok?(states′,
                              memory_write_transformers(pm′, address_block(waddr, length(bl′))))
               ∧ states′(s)
         ⊃
         data(memory_read_list_nse(pm′)(waddr, length(bl′))
                                   (state(memory_write_list_nse(pm′)(waddr, bl′)(s))))
          = bl′
```

Hiding formulas: 2,

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_read_list_write_list_nse_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_nse.4`.

Q.E.D.

C.123.42 Memory_Change_2.unchanged_memory_read_list_write_list_TCC1

Terse proof for `unchanged_memory_read_list_write_list_TCC1`.

`unchanged_memory_read_list_write_list_TCC1`:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
s: State):
unchanged_memory_invariant?(pm, states,
(memory_read_transformers(pm, address_block(waddr, length
address_block(waddr, length(bl)))
∧
unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
transformers_ok?(states,
(memory_read_transformers(pm, address_block(waddr, length(bl))) ∪ m
∧
transformers_ok?(states,
(memory_write_transformers(pm, address_block(waddr, length(bl))) ∪
∧
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
memory_read_side_effect(pm))
∧
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
memory_write_side_effect(pm))
∧ states(s)
⊃
OK?[State, Unit](memory_write_list[State](pm)(waddr, bl)(s)) ∨
Exception?[State, Unit](memory_write_list[State](pm)(waddr, bl)(s))

Repeatedly Skolemizing and flattening,

Using lemma memory_write_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right transformer_invariant_union_transformers unchanged_memory_write_invariant_transformer_invariant

Rewriting using transformer_invariant_union_transformers, matching in *,

Hiding formulas: -1, 2,

Forward chaining on unchanged_memory_invariant_invariant,

Hiding formulas: -2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_TCC1`.

Q.E.D.

C.123.43 Memory_Change_2.unchanged_memory_read_list_write_list_TCC2

Terse proof for `unchanged_memory_read_list_write_list_TCC2`.

unchanged_memory_read_list_write_list_TCC2:

{1} ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
s: State):
unchanged_memory_invariant?(pm, states,
(memory_read_transformers(pm, address_block(waddr, length(bl))) ∪ (me
address_block(waddr, length(bl)))
∧
unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
transformers_ok?(states,
(memory_read_transformers(pm, address_block(waddr, length(bl))) ∪ memory_read
∧
transformers_ok?(states,
(memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ memory_w
∧
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
memory_read_side_effect(pm))
∧
side_effect_content_unchanged(address_block(waddr, length(bl)), states,
memory_write_side_effect(pm))
∧ states(s)
⊃
OK?[State, list[Byte]]
(memory_read_list[State]
(pm)(waddr, length[Byte](bl))
(state[State, Unit](memory_write_list[State](pm)(waddr, bl)(s))))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right transformer_invariant_union_transformers has_next_state memory_write_side_effect_super_transformers_memory_write_side_effect

Forward chaining on unchanged_memory_invariant_invariant,

Using lemma memory_read_list_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

unchanged_memory_read_list_write_list_TCC2.1:

```
{-1}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-2}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′)))
{-3}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length
                                  address_block(waddr′, length(bl′)))
{-4}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-5}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-6}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-7}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
{-8}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-9}  states′(s′)
------------------------------------------------------------------------------
{1}   states′(state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′)))
{2}   OK?[State, list[Byte]]
          (memory_read_list[State]
               (pm′)(waddr′, length[Byte](bl′))
               (state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′))))
```

Hiding formulas: 2,
Using lemma unchanged_memory_write_invariant_transformer_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma memory_write_list_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

unchanged_memory_read_list_write_list_TCC2.1.1:

```
{-1}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ me
{-2}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-3}  states′(s′)
{-4}  OK?(memory_write_list(pm′)(waddr′, bl′)(s′))
{-5}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-6}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       ad-
      dress_block(waddr′, length(bl′))))
{-7}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-8}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, lengt
                                  address_block(waddr′, length(bl′)))
{-10} changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-11} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
------------------------------------------------------------------------------
{1}   states′(state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′)))
```

Using lemma transformer_invariant_write_list,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_read_list_write_list_TCC2.1.1.1:`

{-1} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′, memory_write_side_effect(pm′))
{-2} states′(s′)
{-3} result_pred(states′)(expr_2_super(memory_write_list(pm′)(waddr′, bl′))(s′))
{-4} transformers_ok?(states′, (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write
{-5} OK?(memory_write_list(pm′)(waddr′, bl′)(s′))
{-6} unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-7} transformer_invariant?(states′, memory_write_transformers(pm′, address_block(waddr′, length(bl′))))
{-8} transformers_ok?(states′, (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_read_s
{-9} transformer_invariant?(states′, (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (memor
{-10} unchanged_memory_invariant?(pm′, states′, (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (r address_block(waddr′, length(bl′))))
{-11} changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-12} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′, memory_read_side_effect(pm′))

{1} states′(state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′)))

Rewriting using expr_result_pred_next_state, matching in *,

This completes the proof of `unchanged_memory_read_list_write_list_TCC2.1.1.1`.

`unchanged_memory_read_list_write_list_TCC2.1.1.2:`

```
{-1}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-2}  states′(s′)
{-3}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ me
{-4}  OK?(memory_write_list(pm′)(waddr′, bl′)(s′))
{-5}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-6}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       ad-
      dress_block(waddr′, length(bl′))))
{-7}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mer
{-8}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, lengt
                                  address_block(waddr′, length(bl′)))
{-10} changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-11} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
-------------------------------------------------------------------------------
 {1}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′)
 {2}  states′(state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′)))
```

Rewriting using add_as_union, matching in *,

Rewriting using transformer_invariant_union_transformers, matching in *,

Keeping (-8 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_TCC2.1.1.2`.

`unchanged_memory_read_list_write_list_TCC2.1.2:`

```
{-1}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write
{-2}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-3}  states′(s′)
{-4}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-5}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       ad-
      dress_block(waddr′, length(bl′))))
{-6}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_read_s
{-7}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (memor
{-8}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (r
                                  address_block(waddr′, length(bl′)))
{-9}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-10} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
-------------------------------------------------------------------------------
 {1}  transformer_invariant?(states′,
                             (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memor
 {2}  states′(state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′)))
```

Keeping (-5 -7 1) and hiding *,

Rewriting using transformer_invariant_union_transformers, matching in *,

Hiding formulas: -1, 2,

Using lemma transformer_invariant_mono_transformers,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `unchanged_memory_read_list_write_list_TCC2.1.2`.

`unchanged_memory_read_list_write_list_TCC2.2:`

```
{-1}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-2}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′)))
{-3}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length
                                  address_block(waddr′, length(bl′)))
{-4}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-5}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-6}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-7}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
{-8}  side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-9}  states′(s′)
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length [Byt
{2}   OK?[State, list[Byte]]
          (memory_read_list[State]
                (pm′)(waddr′, length[Byte](bl′))
                (state[State, Unit](memory_write_list[State](pm′)(waddr′, bl′)(s′))))
```

Keeping (-2 1) and hiding *,
Rewriting using transformer_invariant_union_transformers, matching in *,
we get 2 subgoals:
`unchanged_memory_read_list_write_list_TCC2.2.1:`

```
{-1}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′)))
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_read_transformers(pm′,
                                                      address_block(waddr′,
                                                                    length[Byte](bl′))))
{2}   transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length [Byt
```

Hiding formulas: 2,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_read_list_write_list_TCC2.2.1`.
`unchanged_memory_read_list_write_list_TCC2.2.2:`

```
{-1}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′)))
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_read_side_effect_super_transformers(pm′,
                                                                        ad-
      dress_block
                                                                        (waddr′,
                                                                          length[Byte](b
{2}   transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length [Byt
```

Hiding formulas: 2,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `unchanged_memory_read_list_write_list_TCC2.2.2`.
Q.E.D.

C.123.44 Memory_Change_2.unchanged_memory_read_list_write_list

Terse proof for `unchanged_memory_read_list_write_list`.

`unchanged_memory_read_list_write_list`:

```
{1}  ∀ (pm: Memory_struct[State], states: PRED[State], waddr: Address, bl: list[Byte],
        s: State):
       unchanged_memory_invariant?(pm, states,
                                   (memory_read_transformers(pm, address_block(waddr, length(bl))) ∪ (me
                                   address_block(waddr, length(bl)))
        ∧
        unchanged_memory_write_invariant?(pm, states, address_block(waddr, length(bl))) ∧
         changed_memory_invariant?(pm, states, address_block(waddr, length(bl))) ∧
          transformers_ok?(states,
                           (memory_read_transformers(pm, address_block(waddr, length(bl))) ∪ memory_read
           ∧
           transformers_ok?(states,
                            (memory_write_transformers(pm, address_block(waddr, length(bl))) ∪ memory_w
            ∧
            side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                          memory_read_side_effect(pm))
             ∧
             side_effect_content_unchanged(address_block(waddr, length(bl)), states,
                                           memory_write_side_effect(pm))
              ∧ states(s)
        ⊃
        data(memory_read_list(pm)(waddr, length(bl))
                          (state(memory_write_list(pm)(waddr, bl)(s))))
         = bl
```

Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_read_list_write_list_TCC2,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: address_block address_block_subset_1 ##! length subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right memory_read_transformers_memory_read memory_read_transformers_mono memory_write_transformers_memory_write memory_write_transformers_mono transformer_invariant_mono_transformers expr_transformer_invariant_next_ok ok_result has_next_state state_expr_2_super ok_expr_2_super memory_write_side_effect_super_transformers_memory_write_side_effect memory_read_side_effect_super_transformers_memory_read_side_effect
Using lemma expr_transformers_ok_ok,
Simplifying, rewriting, and recording with decision procedures,
Forward chaining on unchanged_memory_invariant_invariant,
Using lemma expr_transformer_invariant_next_ok,
Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of memory_read_list,

Expanding the definition of memory_write_list,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma side_effect_content_unchanged_content,

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -1,

Using lemma unchanged_memory_read_list_write_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 4 subgoals:

unchanged_memory_read_list_write_list.1:

{-1} unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-2} changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3} states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-4} data(memory_read_list_nse(pm′)(waddr′, length(bl′))
(state(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)
(s′))))))
= bl′
{-5} data(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)) = bl′
{-6} transformer_invariant?(states′,
(memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (memor
{-7} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-8} OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)
(s′))))))
{-9} OK?[State, list[Byte]]
(memory_read_side_effect(pm′)
(waddr′,
data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′, bl′)
(state
(memory_write_side_effect
(pm′)
(waddr′, bl′, FALSE)
(s′))))),
FALSE)
(state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′, bl′)
(state
(memory_write_side_effec
(pm′)
(waddr′, bl′, FALSE)
(s′))))))))
{-10} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (r
address_block(waddr′, length(bl′)))
{-11} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_read_s
{-12} transformers_ok?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write
{-13} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_read_side_effect(pm′))
{-14} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
memory_write_side_effect(pm′))
{-15} states′(s′)

{1} data(memory_read_side_effect(pm′)
(waddr′,
data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
(state[State, Unit]
(memory_write_list_nse(pm′)
(waddr′, bl′)
(state
(memory_write_side_effect
(pm′)
(waddr′, bl′, FALSE)

Replacing using formula -4,

Using lemma side_effect_content_unchanged_content,

Simplifying, rewriting, and recording with decision procedures,

Hiding formulas: -4, -5, -9, -13, -14, 2,

Using lemma memory_read_list_nse_next_length,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`unchanged_memory_read_list_write_list.1.1:`

```
{-1}  unchanged_memory_write_invariant?(pm', states', address_block(waddr', length(bl')))
{-2}  changed_memory_invariant?(pm', states', address_block(waddr', length(bl')))
{-3}  states'(state(memory_write_side_effect(pm')(waddr', bl', FALSE)(s')))
{-4}  transformer_invariant?(states',
                             (memory_read_transformers(pm', address_block(waddr', length(bl'))
{-5}  OK?(memory_write_side_effect(pm')(waddr', bl', FALSE)(s'))
{-6}  OK?(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
                          (state[State, Unit]
                             (memory_write_list_nse(pm')(waddr', bl')
                                                        (state(memory_write_side_effect(p
                                                               (waddr', bl', FALSE)
                                                               (s'))))))
{-7}  unchanged_memory_invariant?(pm', states',
                                  (memory_read_transformers(pm', address_block(waddr', lengt
                                  address_block(waddr', length(bl')))
{-8}  transformers_ok?(states',
                       (memory_read_transformers(pm', address_block(waddr', length(bl'))) ∪ mer
{-9}  transformers_ok?(states',
                       (memory_write_transformers(pm', address_block(waddr', length(bl'))) ∪ me
{-10} states'(s')
-------------------------------------------------------------------------------------------------
 {1}  states'(state[State, Unit]
                 (memory_write_list_nse(pm')(waddr', bl')
                                            (state(memory_write_side_effect(pm')
                                                   (waddr', bl', FALSE)(s')))))
 {2}  states'(state(memory_read_list_nse(pm')(waddr', length[Byte](bl'))
                                            (state[State, Unit]
                                               (memory_write_list_nse(pm')
                                                                     (waddr', bl')
                                                                     (state
                                                                       (memory_write_s
                                                                         (pm')
                                                                         (waddr', bl',
                                                                         (s')))))))
```

Hiding formulas: 2,

Using lemma transformer_invariant_write_list_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_read_list_write_list.1.1.1:`

{-1} states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-2} result_pred(states′)
(expr_2_super(memory_write_list_nse(pm′)(waddr′, bl′))
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′))))
{-3} unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-4} changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-5} transformer_invariant?(states′,
(memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (memor
{-6} OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-7} OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)
(s′))))))
{-8} unchanged_memory_invariant?(pm′, states′,
(memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (r
address_block(waddr′, length(bl′)))
{-9} transformers_ok?(states′,
(memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_read_s
{-10} transformers_ok?(states′,
(memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write
{-11} states′(s′)

{1} states′(state[State, Unit]
(memory_write_list_nse(pm′)(waddr′, bl′)
(state(memory_write_side_effect(pm′)
(waddr′, bl′, FALSE)(s′)))))

Using lemma expr_result_pred_next_state[State, Unit],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: -1, -7, 2,

Using lemma memory_write_list_ok_nse,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`unchanged_memory_read_list_write_list.1.1.1.1:`

```
{-1}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-2}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-4}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′)))
{-5}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-6}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length
                                  address_block(waddr′, length(bl′)))
{-7}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-8}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-9}  states′(s′)
-------------------------------------------------------------------------------
{1}   transformers_ok?(states′,
                       memory_write_transformers(pm′, address_block(waddr′, length(bl′))))
{2}   has_next_state(memory_write_list_nse(pm′)(waddr′, bl′)
                                                (state(memory_write_side_effect(pm′)
                                                       (waddr′, bl′, FALSE)(s′))))
```

Keeping (-8 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list.1.1.1.1`.

`unchanged_memory_read_list_write_list.1.1.1.2:`

```
{-1}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-2}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-4}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′)))
{-5}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-6}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length
                                  address_block(waddr′, length(bl′)))
{-7}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-8}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-9}  states′(s′)
-------------------------------------------------------------------------------
{1}   transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       ad-
      dress_block(waddr′, length(bl′))))
{2}   has_next_state(memory_write_list_nse(pm′)(waddr′, bl′)
                                                (state(memory_write_side_effect(pm′)
                                                       (waddr′, bl′, FALSE)(s′))))
```

Forward chaining on unchanged_memory_write_invariant_transformer_invariant,

This completes the proof of `unchanged_memory_read_list_write_list.1.1.1.2`.

`unchanged_memory_read_list_write_list.1.1.2:`

```
{-1}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-2}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-4}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (memor
{-5}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-6}  OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                           (state[State, Unit]
                                  (memory_write_list_nse(pm′)(waddr′, bl′)
                                                        (state(memory_write_side_effect(pm′)
                                                                                       (waddr′, bl′, FALSE)
                                                                                       (s′))))))
{-7}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (r
                                  address_block(waddr′, length(bl′)))
{-8}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_read_s
{-9}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write
{-10} states′(s′)
------------------------------------------------------------------------------------------------------------
 {1}  transformer_invariant?(states′,
                             memory_write_transformers(pm′,
                                                       ad-
      dress_block(waddr′, length(bl′))))
 {2}  states′(state[State, Unit]
                  (memory_write_list_nse(pm′)(waddr′, bl′)
                                             (state(memory_write_side_effect(pm′)
                                                                            (waddr′, bl′, FALSE)(s′)))))
```

Forward chaining on unchanged_memory_write_invariant_transformer_invariant,

This completes the proof of `unchanged_memory_read_list_write_list.1.1.2`.

unchanged_memory_read_list_write_list.1.2:

```
{-1}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-2}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-4}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))
{-5}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-6}  OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                           (state[State, Unit]
                              (memory_write_list_nse(pm′)(waddr′, bl′)
                                                    (state(memory_write_side_effect(p
                                                           (waddr′, bl′, FALSE)
                                                           (s′))))))
{-7}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, lengt
                                  address_block(waddr′, length(bl′)))
{-8}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mer
{-9}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ me
{-10} states′(s′)
-------------------------------------------------------------------------------------------------
 {1}  transformers_ok?(states′,
                       memory_read_transformers(pm′,
                                                address_block(waddr′, length[Byte](bl′))))
 {2}  states′(state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                             (state[State, Unit]
                                                (memory_write_list_nse(pm′)
                                                                      (waddr′, bl′)
                                                                      (state
                                                                        (memory_write_s
                                                                          (pm′)
                                                                          (waddr′, bl′,
                                                                          (s′)))))))
```

Keeping (-8 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of unchanged_memory_read_list_write_list.1.2.

`unchanged_memory_read_list_write_list.1.3:`

```
{-1}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-2}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-4}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (memor
{-5}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-6}  OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                             (state[State, Unit]
                                (memory_write_list_nse(pm′)(waddr′, bl′)
                                                        (state(memory_write_side_effect(pm′)
                                                                (waddr′, bl′, FALSE)
                                                                (s′))))))
{-7}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ (r
                                  address_block(waddr′, length(bl′)))
{-8}  transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_read_s
{-9}  transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ memory_write
{-10} states′(s′)
-------------------------------------------------------------------------------------------------------
 {1}  transformer_invariant?(states′,
                             memory_read_transformers(pm′,
                                                      address_block(waddr′,
                                                                    length[Byte](bl′))))
 {2}  states′(state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                          (state[State, Unit]
                                             (memory_write_list_nse(pm′)
                                                                   (waddr′, bl′)
                                                                   (state
                                                                      (memory_write_side_effect
                                                                        (pm′)
                                                                        (waddr′, bl′, FALSE)
                                                                        (s′)))))))
```

Keeping (-4 1) and hiding *,

Using lemma transformer_invariant_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list.1.3`.

```
unchanged_memory_read_list_write_list.2:

{-1}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-2}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-4}  data(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)) = bl′
{-5}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))
{-6}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-7}  OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                   (state[State, Unit]
                                          (memory_write_list_nse(pm′)(waddr′, bl′)
                                                                (state(memory_write_side_effect(p
                                                                       (waddr′, bl′, FALSE)
                                                                       (s′))))))
{-8}  OK?[State, list[Byte]]
         (memory_read_side_effect(pm′)
                 (waddr′,
                   data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                                 (state[State, Unit]
                                                        (memory_write_list_nse(pm′)
                                                                              (waddr′, bl′)
                                                                              (state
                                                                                (memory_write
                                                                                  (pm′)
                                                                                  (waddr′, bl′
                                                                                  (s′))))))),
                   FALSE)
                 (state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                                 (state[State, Unit]
                                                        (memory_write_list_nse(pm′)
                                                                              (waddr′, bl′)
                                                                              (state
                                                                                (memory_writ
                                                                                  (pm′)
                                                                                  (waddr′, bl
                                                                                  (s′)))))))))
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, lengt
                                  address_block(waddr′, length(bl′))))
{-10} transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-11} transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ me
{-12} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
{-13} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-14} states′(s′)
------------------------------------------------------------------------------
 {1}  transformers_ok?(states′,
                       memory_write_transformers(pm′, address_block(waddr′, length(bl′))))
 {2}  data(memory_read_side_effect(pm′)
                 (waddr′,
                   data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                                 (state[State, Unit]
                                                        (memory_write_list_nse(pm′)
                                                                              (waddr′, bl′)
                                                                              (state
                                                                                (memory_write
                                                                                  (pm′)
                                                                                  (waddr′, bl′
                                                                                  (s′))))))),
                   FALSE)
                 (state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                                 (state[State, Unit]
```

Keeping (-11 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list.2`.

```
unchanged_memory_read_list_write_list.3:

{-1}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-2}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-4}  data(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)) = bl′
{-5}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))
{-6}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-7}  OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                           (state[State, Unit]
                                 (memory_write_list_nse(pm′)(waddr′, bl′)
                                                       (state(memory_write_side_effect(p
                                                             (waddr′, bl′, FALSE)
                                                             (s′))))))
{-8}  OK?[State, list[Byte]]
        (memory_read_side_effect(pm′)
                (waddr′,
                  data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                           (state[State, Unit]
                                                 (memory_write_list_nse(pm′)
                                                                       (waddr′, bl′)
                                                                       (state
                                                                         (memory_write
                                                                           (pm′)
                                                                           (waddr′, bl′
                                                                           (s′))))))),
                  FALSE)
                (state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                           (state[State, Unit]
                                                 (memory_write_list_nse(pm′)
                                                                       (waddr′, bl′)
                                                                       (state
                                                                         (memory_writ
                                                                           (pm′)
                                                                           (waddr′, bl
                                                                           (s′))))))))
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, lengt
                                  address_block(waddr′, length(bl′)))
{-10} transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-11} transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ me
{-12} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
{-13} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-14} states′(s′)
-------------------------------------------------------------------------------
 {1}  transformers_ok?(states′,
                       memory_read_transformers(pm′, address_block(waddr′, length(bl′))))
 {2}  data(memory_read_side_effect(pm′)
                (waddr′,
                  data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                           (state[State, Unit]
                                                 (memory_write_list_nse(pm′)
                                                                       (waddr′, bl′)
                                                                       (state
                                                                         (memory_write
                                                                           (pm′)
                                                                           (waddr′, bl′
                                                                           (s′))))))),
                  FALSE)
                (state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                           (state[State, Unit]
```

Keeping (-10 1) and hiding *,

Using lemma transformers_ok_mono_transformers,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `unchanged_memory_read_list_write_list.3`.

```
unchanged_memory_read_list_write_list.4:

{-1}  unchanged_memory_write_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-2}  changed_memory_invariant?(pm′, states′, address_block(waddr′, length(bl′)))
{-3}  states′(state(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)))
{-4}  data(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′)) = bl′
{-5}  transformer_invariant?(states′,
                             (memory_read_transformers(pm′, address_block(waddr′, length(bl′))
{-6}  OK?(memory_write_side_effect(pm′)(waddr′, bl′, FALSE)(s′))
{-7}  OK?(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                           (state[State, Unit]
                                 (memory_write_list_nse(pm′)(waddr′, bl′)
                                                       (state(memory_write_side_effect(p
                                                             (waddr′, bl′, FALSE)
                                                             (s′))))))
{-8}  OK?[State, list[Byte]]
         (memory_read_side_effect(pm′)
                 (waddr′,
                   data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                            (state[State, Unit]
                                                  (memory_write_list_nse(pm′)
                                                                        (waddr′, bl′)
                                                                        (state
                                                                          (memory_write
                                                                            (pm′)
                                                                            (waddr′, bl′
                                                                            (s′))))))),
                   FALSE)
                 (state(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                            (state[State, Unit]
                                                  (memory_write_list_nse(pm′)
                                                                        (waddr′, bl′)
                                                                        (state
                                                                          (memory_writ
                                                                            (pm′)
                                                                            (waddr′, bl
                                                                            (s′)))))))))
{-9}  unchanged_memory_invariant?(pm′, states′,
                                  (memory_read_transformers(pm′, address_block(waddr′, lengt
                                  address_block(waddr′, length(bl′)))
{-10} transformers_ok?(states′,
                       (memory_read_transformers(pm′, address_block(waddr′, length(bl′))) ∪ mem
{-11} transformers_ok?(states′,
                       (memory_write_transformers(pm′, address_block(waddr′, length(bl′))) ∪ me
{-12} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_read_side_effect(pm′))
{-13} side_effect_content_unchanged(address_block(waddr′, length(bl′)), states′,
                                    memory_write_side_effect(pm′))
{-14} states′(s′)
------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm′, states′,
                                  memory_read_transformers(pm′,
                                                           ad-
      dress_block(waddr′, length(bl′))),
                                  address_block(waddr′, length(bl′)))
 {2}  data(memory_read_side_effect(pm′)
                 (waddr′,
                   data(memory_read_list_nse(pm′)(waddr′, length[Byte](bl′))
                                            (state[State, Unit]
                                                  (memory_write_list_nse(pm′)
                                                                        (waddr′, bl′)
                                                                        (state
                                                                          (memory_write
                                                                            (pm′)
                                                                            (waddr′, bl′
                                                                            (s′))))))),
```

Keeping (-9 1) and hiding *,
Using lemma unchanged_memory_invariant_mono,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `unchanged_memory_read_list_write_list.4`.
Q.E.D.

C.124 Proofs for Memory_Change_3 (memory.pvs)

C.124.1 Memory_Change_3.expr_unchanged_memory_invariant_unchanged_TCC1

Terse proof for `expr_unchanged_memory_invariant_unchanged_TCC1`.
`expr_unchanged_memory_invariant_unchanged_TCC1`:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → ExprResult[State, Data]], a: Address):
OK?(memory_read(pm)(a)(s)) ∧
OK?(q(s)) ∧
addresses(a) ∧
transformers(expr_2_super(q)) ∧
states(s) ∧ unchanged_memory_invariant?(pm, states, transformers, addresses)
⊃ OK?[State, Data](q(s)) ∨ Exception?[State, Data](q(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_unchanged_memory_invariant_unchanged_TCC1`.
Q.E.D.

C.124.2 Memory_Change_3.expr_unchanged_memory_invariant_unchanged

Terse proof for `expr_unchanged_memory_invariant_unchanged`.
`expr_unchanged_memory_invariant_unchanged`:

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → ExprResult[State, Data]], a: Address):
unchanged_memory_invariant?(pm, states, transformers, addresses) ∧
states(s) ∧
transformers(expr_2_super(q)) ∧
addresses(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm)(a)(s)) ∧ OK?(memory_read(pm)(a)(state(q(s))))
⊃ data(memory_read(pm)(a)(state(q(s)))) = data(memory_read(pm)(a)(s))

Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_invariant_unchanged,
Installing automatic rewrites from: state_expr_2_super ok_expr_2_super has_next_state
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `expr_unchanged_memory_invariant_unchanged`.

Q.E.D.

C.124.3 Memory_Change_3.expr_unchanged_memory_invariant_next_ok_TCC1

Terse proof for `expr_unchanged_memory_invariant_next_ok_TCC1`.

`expr_unchanged_memory_invariant_next_ok_TCC1:`

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
          transformers: PRED[[State → SuperResult[State]]], s: State,
          q: [State → ExprResult[State, Data]]):
       unchanged_memory_invariant?(pm, states, transformers, addresses) ∧
        states(s) ∧ transformers(expr_2_super(q)) ∧ has_next_state(q(s))
        ⊃ OK?[State, Data](q(s)) ∨ Exception?[State, Data](q(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `expr_unchanged_memory_invariant_next_ok_TCC1`.
Q.E.D.

C.124.4 Memory_Change_3.expr_unchanged_memory_invariant_next_ok

Terse proof for `expr_unchanged_memory_invariant_next_ok`.

`expr_unchanged_memory_invariant_next_ok:`

```
{1}  ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
          transformers: PRED[[State → SuperResult[State]]], s: State,
          q: [State → ExprResult[State, Data]]):
       unchanged_memory_invariant?(pm, states, transformers, addresses) ∧
        states(s) ∧ transformers(expr_2_super(q)) ∧ has_next_state(q(s))
        ⊃ states(state(q(s)))
```

Repeatedly Skolemizing and flattening,
Using lemma unchanged_memory_invariant_next_ok[State],
Installing automatic rewrites from: has_next_state_expr_2_super state_expr_2_super
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `expr_unchanged_memory_invariant_next_ok`.
Q.E.D.

C.125 Proofs for Memory_Change_4 (memory.pvs)

C.125.1 Memory_Change_4.expr_unchanged_memory_invariant_composition

Terse proof for `expr_unchanged_memory_invariant_composition`.

`expr_unchanged_memory_invariant_composition:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State], transformers_1, transformers_2: PRED[[State → SuperResult[State]]], q_1: [State → ExprResult[State, Data1]], q_2: [State → ExprResult[State, Data2]]):
transformers_1(expr_2_super(q_1)) ∧
transformers_2(expr_2_super(q_2)) ∧
transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧
unchanged_memory_invariant?(pm, states, transformers_1, addresses) ∧
unchanged_memory_invariant?(pm, states, transformers_2, addresses)
⊃ unchanged_memory_invariant?(pm, states, singleton(expr_2_super(q_1 ## q_2)), addresses)

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Case splitting on transformer_invariant?(states!1, singleton(expr_2_super(q1!1 ## q2!1))),

we get 2 subgoals:

`expr_unchanged_memory_invariant_composition.1:`

{-1} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))
{-2} transformers′(expr_2_super(q_1'))
{-3} transformers″(expr_2_super(q_2'))
{-4} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5} transformer_invariant?(states′, transformers′)
{-6} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers′(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
{-7} transformer_invariant?(states′, transformers″)
{-8} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers″(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

{1} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2'))) ∧
(∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
singleton(expr_2_super(q_1' ## q_2'))(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) =
data(memory_read(pm′)(a)(s)))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -10,

Instantiating the top quantifier in -6 with the terms: (s!1 expr_2_super(q1!1) a!1),

Installing automatic rewrites from: ok_expr_2_super state_expr_2_super has_next_state has_next_state_expr_2_super ##

Instantiating the top quantifier in -8 with the terms: (state(expr_2_super(q1!1)(s!1)) expr_2_super(q2!1) a!1),

we get 2 subgoals:

`expr_unchanged_memory_invariant_composition.1.1:`

{-1} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))
{-2} transformers′(expr_2_super(q_1'))
{-3} transformers″(expr_2_super(q_2'))
{-4} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5} transformer_invariant?(states′, transformers′)
{-6} states′(s') ∧
transformers′(expr_2_super(q_1')) ∧
addresses′(a') ∧
OK?(expr_2_super(q_1')(s')) ∧
OK?(memory_read(pm′)(a')(s')) ∧
OK?(memory_read(pm′)(a')(state(expr_2_super(q_1')(s'))))
⊃
data(memory_read(pm′)(a')(state(expr_2_super(q_1')(s')))) =
data(memory_read(pm′)(a')(s'))
{-7} transformer_invariant?(states′, transformers″)
{-8} states′(state(expr_2_super(q_1')(s'))) ∧
transformers″(expr_2_super(q_2')) ∧
addresses′(a') ∧
OK?(expr_2_super(q_2')(state(expr_2_super(q_1')(s')))) ∧
OK?(memory_read(pm′)(a')(state(expr_2_super(q_1')(s')))) ∧
OK?(memory_read(pm′)
(a')(state(expr_2_super(q_2')(state(expr_2_super(q_1')(s'))))))
⊃
data(memory_read(pm′)
(a')(state(expr_2_super(q_2')(state(expr_2_super(q_1')(s'))))))
= data(memory_read(pm′)(a')(state(expr_2_super(q_1')(s'))))
{-9} states′(s')
{-10} q' = expr_2_super(q_1' ## q_2')
{-11} addresses′(a')
{-12} OK?(expr_2_super(q_1' ## q_2')(s'))
{-13} OK?(memory_read(pm′)(a')(s'))
{-14} OK?(memory_read(pm′)(a')(state(expr_2_super(q_1' ## q_2')(s'))))

{1} data(memory_read(pm′)(a')(state(expr_2_super(q_1' ## q_2')(s')))) =
data(memory_read(pm′)(a')(s'))

Expanding the definition of ##,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`expr_unchanged_memory_invariant_composition.1.1.1:`

```
{-1}  transformer_invariant?(states′,
                             singleton(expr_2_super(λ (s: State):
                                                      CASES q′1(s) OF
                                                        OK(state, data): q′2(state),
                                                        Excep-
      tion(ex_type, state):
                                                          Excep-
      tion(ex_type, state),
                                                        Fatal: Fatal,
                                                        Hang: Hang
                                                        ENDCASES)))
{-2}  transformers′(expr_2_super(q′1))
{-3}  transformers″(expr_2_super(q′2))
{-4}  transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5}  transformer_invariant?(states′, transformers′)
{-6}  states′(s′)
{-7}  addresses′(a′)
{-8}  OK?(memory_read(pm′)(a′)(s′))
{-9}  data(memory_read(pm′)(a′)(state(q′1(s′)))) =
       data(memory_read(pm′)(a′)(s′))
{-10} transformer_invariant?(states′, transformers″)
{-11} q′ =
       expr_2_super(λ (s: State):
                      CASES q′1(s) OF
                        OK(state, data): q′2(state),
                        Exception(ex_type, state): Exception(ex_type, state),
                        Fatal: Fatal,
                        Hang: Hang
                        ENDCASES)
{-12} OK?(q′1(s′))
{-13} OK?(q′2(state(q′1(s′))))
{-14} OK?(memory_read(pm′)(a′)(state(q′2(state(q′1(s′)))))))
------------------------------------------------------------
 {1}  states′(state(q′1(s′)))
 {2}  data(memory_read(pm′)(a′)(state(q′2(state(q′1(s′)))))) =
       data(memory_read(pm′)(a′)(s′))
```

Rewriting using expr_transformer_invariant_next_ok, matching in * where s gets s!1, q gets q1!1, states gets states!1, transformers gets transformers!1,

This completes the proof of `expr_unchanged_memory_invariant_composition.1.1.1`.

expr_unchanged_memory_invariant_composition.1.1.2:

```
{-1}  transformer_invariant?(states′,
                             singleton(expr_2_super(λ (s: State):
                                                      CASES q′1(s) OF
                                                        OK(state, data): q′2(state),
                                                        Excep-
      tion(ex_type, state):
                                                          Excep-
      tion(ex_type, state),
                                                        Fatal: Fatal,
                                                        Hang: Hang
                                                        ENDCASES)))
{-2}  transformers′(expr_2_super(q′1))
{-3}  transformers″(expr_2_super(q′2))
{-4}  transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5}  transformer_invariant?(states′, transformers′)
{-6}  states′(s′)
{-7}  addresses′(a′)
{-8}  OK?(memory_read(pm′)(a′)(s′))
{-9}  transformer_invariant?(states′, transformers″)
{-10} q′ =
       expr_2_super(λ (s: State):
                      CASES q′1(s) OF
                        OK(state, data): q′2(state),
                        Exception(ex_type, state): Exception(ex_type, state),
                        Fatal: Fatal,
                        Hang: Hang
                        ENDCASES)
{-11} OK?(q′1(s′))
{-12} OK?(q′2(state(q′1(s′))))
{-13} OK?(memory_read(pm′)(a′)(state(q′2(state(q′1(s′))))))
-------------------------------------------------------------
 {1}  OK?(memory_read(pm′)(a′)(state(q′1(s′))))
 {2}  data(memory_read(pm′)(a′)(state(q′2(state(q′1(s′)))))) =
       data(memory_read(pm′)(a′)(s′))
```

Rewriting using expr_transformers_ok_ok, matching in * where s gets state(q1!1(s!1)), q gets memory_read(pm!1)(a!1), states gets states!1, transformers gets memory_read_transformers(pm!1, addresses!1),

we get 2 subgoals:

`expr_unchanged_memory_invariant_composition.1.1.2.1:`

```
{-1}  transformer_invariant?(states′,
                             singleton(expr_2_super(λ (s: State):
                                                      CASES q′1(s) OF
                                                        OK(state, data): q′2(state),
                                                        Excep-
      tion(ex_type, state):
                                                          Excep-
      tion(ex_type, state),
                                                        Fatal: Fatal,
                                                        Hang: Hang
                                                      ENDCASES)))
{-2}  transformers′(expr_2_super(q′1))
{-3}  transformers″(expr_2_super(q′2))
{-4}  transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5}  transformer_invariant?(states′, transformers′)
{-6}  states′(s′)
{-7}  addresses′(a′)
{-8}  OK?(memory_read(pm′)(a′)(s′))
{-9}  transformer_invariant?(states′, transformers″)
{-10} q′ =
       expr_2_super(λ (s: State):
                      CASES q′1(s) OF
                        OK(state, data): q′2(state),
                        Exception(ex_type, state): Exception(ex_type, state),
                        Fatal: Fatal,
                        Hang: Hang
                        ENDCASES)
{-11} OK?(q′1(s′))
{-12} OK?(q′2(state(q′1(s′))))
{-13} OK?(memory_read(pm′)(a′)(state(q′2(state(q′1(s′))))))
-------------------------------------------------------------
 {1}  states′(state(q′1(s′)))
 {2}  OK?(memory_read(pm′)(a′)(state(q′1(s′))))
 {3}  data(memory_read(pm′)(a′)(state(q′2(state(q′1(s′)))))) =
       data(memory_read(pm′)(a′)(s′))
```

Rewriting using expr_transformer_invariant_next_ok, matching in * where s gets s!1, q gets q1!1, states gets states!1, transformers gets transformers!1,

This completes the proof of `expr_unchanged_memory_invariant_composition.1.1.2.1`.

`expr_unchanged_memory_invariant_composition.1.1.2.2:`

```
{-1}  transformer_invariant?(states',
                             singleton(expr_2_super(λ (s: State):
                                                      CASES q1'(s) OF
                                                        OK(state, data): q2'(state),
                                                        Excep-
      tion(ex_type, state):
                                                          Excep-
      tion(ex_type, state),
                                                        Fatal: Fatal,
                                                        Hang: Hang
                                                        ENDCASES)))
{-2}  transformers'(expr_2_super(q1'))
{-3}  transformers''(expr_2_super(q2'))
{-4}  transformers_ok?(states', memory_read_transformers(pm', addresses'))
{-5}  transformer_invariant?(states', transformers')
{-6}  states'(s')
{-7}  addresses'(a')
{-8}  OK?(memory_read(pm')(a')(s'))
{-9}  transformer_invariant?(states', transformers'')
{-10} q' =
       expr_2_super(λ (s: State):
                      CASES q1'(s) OF
                        OK(state, data): q2'(state),
                        Exception(ex_type, state): Exception(ex_type, state),
                        Fatal: Fatal,
                        Hang: Hang
                        ENDCASES)
{-11} OK?(q1'(s'))
{-12} OK?(q2'(state(q1'(s'))))
{-13} OK?(memory_read(pm')(a')(state(q2'(state(q1'(s'))))))
-------------------------------------------------------------------------------
 {1}  memory_read_transformers(pm', addresses')(expr_2_super(memory_read(pm')(a')))
 {2}  OK?(memory_read(pm')(a')(state(q1'(s'))))
 {3}  data(memory_read(pm')(a')(state(q2'(state(q1'(s')))))) =
       data(memory_read(pm')(a')(s'))
```

Rewriting using memory_read_transformers_memory_read, matching in *,

This completes the proof of `expr_unchanged_memory_invariant_composition.1.1.2.2`.

`expr_unchanged_memory_invariant_composition.1.2:`

{-1}	transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))
{-2}	transformers′(expr_2_super(q_1'))
{-3}	transformers″(expr_2_super(q_2'))
{-4}	transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-5}	transformer_invariant?(states′, transformers′)
{-6}	states′(s') ∧ transformers′(expr_2_super(q_1')) ∧ addresses′(a') ∧ OK?(expr_2_super(q_1')(s')) ∧ OK?(memory_read(pm′)(a')(s')) ∧ OK?(memory_read(pm′)(a')(state(expr_2_super(q_1')(s')))) ⊃ data(memory_read(pm′)(a')(state(expr_2_super(q_1')(s')))) = data(memory_read(pm′)(a')(s'))
{-7}	transformer_invariant?(states′, transformers″)
{-8}	states′(s')
{-9}	q' = expr_2_super(q_1' ## q_2')
{-10}	addresses′(a')
{-11}	OK?(expr_2_super(q_1' ## q_2')(s'))
{-12}	OK?(memory_read(pm′)(a')(s'))
{-13}	OK?(memory_read(pm′)(a')(state(expr_2_super(q_1' ## q_2')(s'))))
{1}	OK?(q_1'(s')) ∨ abnormal?[State](expr_2_super[State, Data1](q_1')(s'))
{2}	data(memory_read(pm′)(a')(state(expr_2_super(q_1' ## q_2')(s')))) = data(memory_read(pm′)(a')(s'))

Keeping (-11 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `expr_unchanged_memory_invariant_composition.1.2`.

`expr_unchanged_memory_invariant_composition.2:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers″(expr_2_super(q_2'))
{-3} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-4} transformer_invariant?(states′, transformers′)
{-5} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers′(q) ∧
addresses′(a) ∧
OK?($q(s)$) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state($q(s)$)))
⊃
data(memory_read(pm′)(a)(state($q(s)$))) = data(memory_read(pm′)(a)(s))
{-6} transformer_invariant?(states′, transformers″)
{-7} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers″(q) ∧
addresses′(a) ∧
OK?($q(s)$) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state($q(s)$)))
⊃
data(memory_read(pm′)(a)(state($q(s)$))) = data(memory_read(pm′)(a)(s))

{1} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))
{2} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2'))) ∧
(∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
singleton(expr_2_super(q_1' ## q_2'))(q) ∧
addresses′(a) ∧
OK?($q(s)$) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state($q(s)$)))
⊃
data(memory_read(pm′)(a)(state($q(s)$))) =
data(memory_read(pm′)(a)(s)))

Rewriting using expr_composition_transformer_invariant, matching in *,

we get 2 subgoals:

`expr_unchanged_memory_invariant_composition.2.1:`

{-1} $\text{transformers}'(\text{expr\_2\_super}(q_1'))$
{-2} $\text{transformers}''(\text{expr\_2\_super}(q_2'))$
{-3} $\text{transformers\_ok?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{addresses}'))$
{-4} $\text{transformer\_invariant?}(\text{states}', \text{transformers}')$
{-5} $\forall\ (s\colon \text{State}, q\colon [\text{State} \to \text{SuperResult}[\text{State}]], a\colon \text{Address})\colon$
$\text{states}'(s) \wedge$
$\text{transformers}'(q) \wedge$
$\text{addresses}'(a) \wedge$
$\text{OK?}(q(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) = \text{data}(\text{memory\_read}(\text{pm}')(a)(s))$
{-6} $\text{transformer\_invariant?}(\text{states}', \text{transformers}'')$
{-7} $\forall\ (s\colon \text{State}, q\colon [\text{State} \to \text{SuperResult}[\text{State}]], a\colon \text{Address})\colon$
$\text{states}'(s) \wedge$
$\text{transformers}''(q) \wedge$
$\text{addresses}'(a) \wedge$
$\text{OK?}(q(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) = \text{data}(\text{memory\_read}(\text{pm}')(a)(s))$

{1} $\text{transformer\_invariant?}(\text{states}', \text{singleton}(\text{expr\_2\_super}(q_1')))$
{2} $\text{transformer\_invariant?}(\text{states}', \text{singleton}(\text{expr\_2\_super}(q_1' \mathbin{\#\#} q_2')))$
{3} FALSE $\wedge$
$(\forall\ (s\colon \text{State}, q\colon [\text{State} \to \text{SuperResult}[\text{State}]], a\colon \text{Address})\colon$
$\text{states}'(s) \wedge$
$\text{singleton}(\text{expr\_2\_super}(q_1' \mathbin{\#\#} q_2'))(q) \wedge$
$\text{addresses}'(a) \wedge$
$\text{OK?}(q(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(\text{state}(q(s)))) =$
$\text{data}(\text{memory\_read}(\text{pm}')(a)(s)))$

Keeping (-1 -3 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(q1!1)), transformers_2 gets transformers!1,

Hiding formulas: (-2 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `expr_unchanged_memory_invariant_composition.2.1`.

`expr_unchanged_memory_invariant_composition.2.2:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers″(expr_2_super(q_2'))
{-3} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-4} transformer_invariant?(states′, transformers′)
{-5} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers′(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))
{-6} transformer_invariant?(states′, transformers″)
{-7} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
transformers″(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) = data(memory_read(pm′)(a)(s))

{1} transformer_invariant?(states′, singleton(expr_2_super(q_2')))
{2} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))
{3} FALSE ∧
(∀ (s: State, q: [State → SuperResult[State]], a: Address):
states′(s) ∧
singleton(expr_2_super(q_1' ## q_2'))(q) ∧
addresses′(a) ∧
OK?(q(s)) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state(q(s))))
⊃
data(memory_read(pm′)(a)(state(q(s)))) =
data(memory_read(pm′)(a)(s)))

Keeping (-2 -5 1) and hiding *,

Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets singleton(expr_2_super(q2!1)), transformers_2 gets transformers!2,

Hiding formulas: (-2 2),

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `expr_unchanged_memory_invariant_composition.2.2`.

Q.E.D.

C.125.2 Memory_Change_4.fexpr_unchanged_memory_invariant_composition

Terse proof for `fexpr_unchanged_memory_invariant_composition`.

`fexpr_unchanged_memory_invariant_composition:`

{1} ∀ (addresses: PRED[Address], pm: Memory_struct[State], states: PRED[State],
transformers_1: PRED[[State → SuperResult[State]]],
q_1: [State → ExprResult[State, Data1]],
q_2: [Data1 → [State → ExprResult[State, Data2]]], P: PRED[Data1]):
(transformers_1(expr_2_super(q_1)) ∧
transformers_ok?(states, memory_read_transformers(pm, addresses)) ∧
unchanged_memory_invariant?(pm, states, transformers_1, addresses) ∧
(∀ (s: (states)): OK?($q_1(s)$) ⊃ P(data($q_1(s)$))) ∧
(∀ (d: (P)):
unchanged_memory_invariant?(pm, states, singleton(expr_2_super($q_2(d)$)),
addresses)))
⊃ unchanged_memory_invariant?(pm, states, singleton(expr_2_super(q_1 ## q_2)), addresses)

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.1:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} ∀ (s: (states′)): OK?($q_1'(s)$) ⊃ P'(data($q_1'(s)$))
{-5} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q_2'(d)$)),
addresses′)

{1} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))

Rewriting using fexpr_composition_transformer_invariant[State, Data1, Data2], matching in * where states gets states!1, P gets P!1,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.1.1:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} ∀ (s: (states′)): OK?($q_1'(s)$) ⊃ P'(data($q_1'(s)$))
{-5} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q_2'(d)$)),
addresses′)

{1} transformer_invariant?(states′, singleton(expr_2_super(q_1')))
{2} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))

Rewriting using transformer_invariant_mono_transformers[State], matching in * where transformers_1 gets singleton(expr_2_super(q1!1)), transformers_2 gets transformers!1,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.1.1.1:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} ∀ (s: (states′)): OK?($q_1'(s)$) ⊃ P'(data($q_1'(s)$))
{-5} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q_2'(d)$)), addresses′)

{1} (singleton(expr_2_super(q_1')) ⊆ transformers′)
{2} transformer_invariant?(states′, singleton(expr_2_super(q_1')))
{3} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))

Keeping (-1 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `fexpr_unchanged_memory_invariant_composition.1.1.1`.

`fexpr_unchanged_memory_invariant_composition.1.1.2:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} ∀ (s: (states′)): OK?($q_1'(s)$) ⊃ P'(data($q_1'(s)$))
{-5} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q_2'(d)$)), addresses′)

{1} transformer_invariant?(states′, transformers′)
{2} transformer_invariant?(states′, singleton(expr_2_super(q_1')))
{3} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))

Expanding the definition of unchanged_memory_invariant?,

which is trivially true.

This completes the proof of `fexpr_unchanged_memory_invariant_composition.1.1.2`.

`fexpr_unchanged_memory_invariant_composition.1.2:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} ∀ (s: (states′)): OK?($q_1'(s)$) ⊃ P'(data($q_1'(s)$))
{-5} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q_2'(d)$)), addresses′)

{1} ∀ (d: (P')): transformer_invariant?(states′, singleton(expr_2_super($q_2'(d)$)))
{2} transformer_invariant?(states′, singleton(expr_2_super(q_1' ## q_2')))

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Expanding the definition of unchanged_memory_invariant?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `fexpr_unchanged_memory_invariant_composition.1.2`.

`fexpr_unchanged_memory_invariant_composition.2:`

{-1} transformers′(expr_2_super(q'_1))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} ∀ (s: (states′)): OK?($q'_1(s)$) ⊃ P'(data($q'_1(s)$))
{-5} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q'_2(d)$)), addresses′)

{1} ∀ (s: State, q: ⌊State → SuperResult⌊State⌋⌋, a: Address):
states′(s) ∧
singleton(expr_2_super(q'_1 ## q'_2))(q) ∧
addresses′(a) ∧
OK?($q(s)$) ∧
OK?(memory_read(pm′)(a)(s)) ∧
OK?(memory_read(pm′)(a)(state($q(s)$)))
⊃
data(memory_read(pm′)(a)(state($q(s)$))) = data(memory_read(pm′)(a)(s))

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -2,

Rewriting using state_expr_2_super, matching in *,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.2.1:`

{-1} states′(s')
{-2} q' = expr_2_super(q'_1 ## q'_2)
{-3} addresses′(a')
{-4} OK?(expr_2_super(q'_1 ## q'_2)(s'))
{-5} OK?(memory_read(pm′)(a')(s'))
{-6} OK?(memory_read(pm′)(a')(state((q'_1 ## q'_2)(s'))))
{-7} transformers′(expr_2_super(q'_1))
{-8} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-9} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-10} ∀ (s: (states′)): OK?($q'_1(s)$) ⊃ P'(data($q'_1(s)$))
{-11} ∀ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q'_2(d)$)), addresses′)

{1} data(memory_read(pm′)(a')(state((q'_1 ## q'_2)(s')))) =
data(memory_read(pm′)(a')(s'))

Rewriting using comp_eval_if_ok_fexpr, matching in *,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.2.1.1:`

{-1} states′(s′)
{-2} q′ = expr_2_super(q′1 ## q′2)
{-3} addresses′(a′)
{-4} OK?(expr_2_super(q′1 ## q′2)(s′))
{-5} OK?(memory_read(pm′)(a′)(s′))
{-6} OK?(memory_read(pm′)(a′)(state((q′1 ## q′2(data(q′1(s′))))(s′))))
{-7} transformers′(expr_2_super(q′1))
{-8} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-9} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-10} ∀ (s: (states′)): OK?(q′1(s)) ⊃ P′(data(q′1(s)))
{-11} ∀ (d: (P′)):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q′2(d))), addresses′)

{1} data(memory_read(pm′)(a′)(state((q′1 ## q′2(data(q′1(s′))))(s′)))) =
data(memory_read(pm′)(a′)(s′))

Instantiating the top quantifier in -11 with the terms: (data(q1!1(s!1))),

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.2.1.1.1:`

{-1} states′(s′)
{-2} q′ = expr_2_super(q′1 ## q′2)
{-3} addresses′(a′)
{-4} OK?(expr_2_super(q′1 ## q′2)(s′))
{-5} OK?(memory_read(pm′)(a′)(s′))
{-6} OK?(memory_read(pm′)(a′)(state((q′1 ## q′2(data(q′1(s′))))(s′))))
{-7} transformers′(expr_2_super(q′1))
{-8} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-9} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-10} ∀ (s: (states′)): OK?(q′1(s)) ⊃ P′(data(q′1(s)))
{-11} unchanged_memory_invariant?(pm′, states′,
singleton(expr_2_super(q′2(data(q′1(s′))))), addresses′)

{1} data(memory_read(pm′)(a′)(state((q′1 ## q′2(data(q′1(s′))))(s′)))) =
data(memory_read(pm′)(a′)(s′))

Using lemma expr_unchanged_memory_invariant_composition,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.2.1.1.1.1:`

{-1}	transformers′(expr_2_super(q_1'))
{-2}	transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3}	unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4}	unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_2'(data(q_1'(s')))))), addresses′)
{-5}	unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_1' ## q_2'(data(q_1'(s'))))), addresses′)
{-6}	states′(s')
{-7}	q' = expr_2_super(q_1' ## q_2')
{-8}	addresses′(a')
{-9}	OK?(expr_2_super(q_1' ## q_2')(s'))
{-10}	OK?(memory_read(pm′)(a')(s'))
{-11}	OK?(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s'))))
{-12}	∀ (s: (states′)): OK?(q_1'(s)) ⊃ P'(data(q_1'(s))))
{1}	data(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s')))) = data(memory_read(pm′)(a')(s'))

Using lemma expr_unchanged_memory_invariant_unchanged,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.1:`

{-1}	unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_1' ## q_2'(data(q_1'(s'))))), addresses′)
{-2}	states′(s')
{-3}	addresses′(a')
{-4}	OK?(memory_read(pm′)(a')(s'))
{-5}	OK?(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s'))))
{-6}	transformers′(expr_2_super(q_1'))
{-7}	transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-8}	unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-9}	unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_2'(data(q_1'(s')))))), addresses′)
{-10}	q' = expr_2_super(q_1' ## q_2')
{-11}	OK?(expr_2_super(q_1' ## q_2')(s'))
{-12}	∀ (s: (states′)): OK?(q_1'(s)) ⊃ P'(data(q_1'(s)))
{1}	OK?((q_1' ## q_2'(data(q_1'(s'))))(s'))
{2}	data(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s')))) = data(memory_read(pm′)(a')(s'))

Keeping (-11 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.1`.

`fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.2:`

{-1} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_1' ## q_2'(data(q_1'(s')))))), addresses′)
{-2} states′(s')
{-3} addresses′(a')
{-4} OK?(memory_read(pm′)(a')(s'))
{-5} OK?(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s'))))
{-6} transformers′(expr_2_super(q_1'))
{-7} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-8} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-9} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_2'(data(q_1'(s'))))), addresses′)
{-10} q' = expr_2_super(q_1' ## q_2')
{-11} OK?(expr_2_super(q_1' ## q_2')(s'))
{-12} ∀ (s: (states′)): OK?(q_1'(s)) ⊃ P'(data(q_1'(s)))

{1} singleton(expr_2_super(q_1' ## q_2'(data(q_1'(s'))))) (expr_2_super((q_1' ## q_2'(data(q_1'(s')))))))
{2} data(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s')))) = data(memory_read(pm′)(a')(s'))

Expanding the definition of singleton,

which is trivially true.

This completes the proof of `fexpr_unchanged_memory_invariant_composition.2.1.1.1.1.2`.

`fexpr_unchanged_memory_invariant_composition.2.1.1.1.2:`

{-1} transformers′(expr_2_super(q_1'))
{-2} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-3} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-4} unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super(q_2'(data(q_1'(s'))))), addresses′)
{-5} states′(s')
{-6} q' = expr_2_super(q_1' ## q_2')
{-7} addresses′(a')
{-8} OK?(expr_2_super(q_1' ## q_2')(s'))
{-9} OK?(memory_read(pm′)(a')(s'))
{-10} OK?(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s'))))
{-11} ∀ (s: (states′)): OK?(q_1'(s)) ⊃ P'(data(q_1'(s)))

{1} singleton(expr_2_super(q_2'(data(q_1'(s'))))) (expr_2_super(q_2'(data(q_1'(s')))))
{2} data(memory_read(pm′)(a')(state((q_1' ## q_2'(data(q_1'(s'))))(s')))) = data(memory_read(pm′)(a')(s'))

Expanding the definition of singleton,

which is trivially true.

This completes the proof of `fexpr_unchanged_memory_invariant_composition.2.1.1.1.2`.

`fexpr_unchanged_memory_invariant_composition.2.1.1.2:`

{-1}	$\text{states}'(s')$
{-2}	$q' = \text{expr\_2\_super}(q'_1 \mathbin{\#\#} q'_2)$
{-3}	$\text{addresses}'(a')$
{-4}	$\text{OK?}(\text{expr\_2\_super}(q'_1 \mathbin{\#\#} q'_2)(s'))$
{-5}	$\text{OK?}(\text{memory\_read}(\text{pm}')(a')(s'))$
{-6}	$\text{OK?}(\text{memory\_read}(\text{pm}')(a')(\text{state}((q'_1 \mathbin{\#\#} q'_2(\text{data}(q'_1(s'))))(s'))))$
{-7}	$\text{transformers}'(\text{expr\_2\_super}(q'_1))$
{-8}	$\text{transformers\_ok?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{addresses}'))$
{-9}	$\text{unchanged\_memory\_invariant?}(\text{pm}', \text{states}', \text{transformers}', \text{addresses}')$
{-10}	$\forall\ (s\colon\ (\text{states}'))\colon\ \text{OK?}(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{1}	$P'(\text{data}\lfloor \text{State}, \text{Data1} \rfloor(q'_1(s')))$
{2}	$\text{data}(\text{memory\_read}(\text{pm}')(a')(\text{state}((q'_1 \mathbin{\#\#} q'_2(\text{data}(q'_1(s'))))(s')))) =$ $\text{data}(\text{memory\_read}(\text{pm}')(a')(s'))$

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-4 3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `fexpr_unchanged_memory_invariant_composition.2.1.1.2`.

`fexpr_unchanged_memory_invariant_composition.2.1.2:`

{-1}	$\text{states}'(s')$
{-2}	$q' = \text{expr\_2\_super}(q'_1 \mathbin{\#\#} q'_2)$
{-3}	$\text{addresses}'(a')$
{-4}	$\text{OK?}(\text{expr\_2\_super}(q'_1 \mathbin{\#\#} q'_2)(s'))$
{-5}	$\text{OK?}(\text{memory\_read}(\text{pm}')(a')(s'))$
{-6}	$\text{OK?}(\text{memory\_read}(\text{pm}')(a')(\text{state}((q'_1 \mathbin{\#\#} q'_2)(s'))))$
{-7}	$\text{transformers}'(\text{expr\_2\_super}(q'_1))$
{-8}	$\text{transformers\_ok?}(\text{states}', \text{memory\_read\_transformers}(\text{pm}', \text{addresses}'))$
{-9}	$\text{unchanged\_memory\_invariant?}(\text{pm}', \text{states}', \text{transformers}', \text{addresses}')$
{-10}	$\forall\ (s\colon\ (\text{states}'))\colon\ \text{OK?}(q'_1(s)) \supset P'(\text{data}(q'_1(s)))$
{-11}	$\forall\ (d\colon\ (P'))\colon$ $\text{unchanged\_memory\_invariant?}(\text{pm}', \text{states}', \text{singleton}(\text{expr\_2\_super}(q'_2(d))),$ $\text{addresses}')$
{1}	$\text{OK?}(q'_1(s'))$
{2}	$\text{data}(\text{memory\_read}(\text{pm}')(a')(\text{state}((q'_1 \mathbin{\#\#} q'_2)(s')))) =$ $\text{data}(\text{memory\_read}(\text{pm}')(a')(s'))$

Keeping (-4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `fexpr_unchanged_memory_invariant_composition.2.1.2`.

`fexpr_unchanged_memory_invariant_composition.2.2:`

{-1} states′(s′)
{-2} $q' = \text{expr\_2\_super}(q'_1 \mathbin{\#\#} q'_2)$
{-3} addresses′(a′)
{-4} OK?(expr_2_super(q'_1 ## q'_2)(s′))
{-5} OK?(memory_read(pm′)(a′)(s′))
{-6} OK?(memory_read(pm′)(a′)(state(expr_2_super(q'_1 ## q'_2)(s′))))
{-7} transformers′(expr_2_super(q'_1))
{-8} transformers_ok?(states′, memory_read_transformers(pm′, addresses′))
{-9} unchanged_memory_invariant?(pm′, states′, transformers′, addresses′)
{-10} $\forall$ (s: (states′)): OK?($q'_1(s)$) $\supset$ P'(data($q'_1(s)$))
{-11} $\forall$ (d: (P')):
unchanged_memory_invariant?(pm′, states′, singleton(expr_2_super($q'_2(d)$)), addresses′)

{1} has_next_state((q'_1 ## q'_2)(s′))
{2} data(memory_read(pm′)(a′)(state(expr_2_super(q'_1 ## q'_2)(s′)))) =
data(memory_read(pm′)(a′)(s′))

Rewriting using ok_expr_2_super, matching in *,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `fexpr_unchanged_memory_invariant_composition.2.2`.
Q.E.D.

C.126 Proofs for Memory_Physical_Memory (physical_memory.pvs)

This theory contains no provable formal statements.

C.127 Proofs for Memory_access (result.pvs)

This theory contains no provable formal statements.

C.128 Proofs for Memory_access_adt (Memory_access_adt.pvs)

This theory contains no provable formal statements.

C.129 Proofs for Memory_access_adt_reduce (Memory_access_adt.pvs)

This theory contains no provable formal statements.

C.130 Proofs for Memory_access_util (result.pvs)

This theory contains no provable formal statements.

C.131 Proofs for Memory_privilege (result.pvs)

This theory contains no provable formal statements.

C.132 Proofs for Memory_privilege_adt (Memory_privilege_adt.pvs)

This theory contains no provable formal statements.

C.133 Proofs for Memory_privilege_adt_reduce (Memory_privilege_adt.pvs)

This theory contains no provable formal statements.

C.134 Proofs for Memory_privilege_util (result.pvs)

C.134.1 Memory_privilege_util.bool_to_memory_privilege_iso

Terse proof for `bool_to_memory_privilege_iso`.

`bool_to_memory_privilege_iso`:

|
|---
| {1} ∀ (ar: Memory_privilege): bool_to_memory_privilege(memory_privilege_to_bool(ar)) = ar

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `bool_to_memory_privilege_iso`.
Q.E.D.

C.135 Proofs for Min_Max_Extension (vfiasco-prelude.pvs)

C.135.1 Min_Max_Extension.max_stable_under_extension_TCC1

Terse proof for `max_stable_under_extension_TCC1`.

`max_stable_under_extension_TCC1`:

|
|---
| {1} ∀ (P: non_empty_finite_set[S]): ¬ empty?[T](extend[T, S, bool, FALSE](P))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `max_stable_under_extension_TCC1`.
Q.E.D.

C.135.2 Min_Max_Extension.max_stable_under_extension

Terse proof for `max_stable_under_extension`.

`max_stable_under_extension:`

```
{1}  ∀ (P: non_empty_finite_set[S]):
       max[T, ≤](extend[T, S, bool, FALSE](P)) =
         max[S, restrict[[T, T], [S, S], boolean](≤)](P)
```

Repeatedly Skolemizing and flattening,
Rewriting using max_lem, matching in *,
Applying propositional simplification,
we get 2 subgoals:
`max_stable_under_extension.1:`

```
{-1}  is_finite[S](P')
{1}   extend[T, S, bool, FALSE]
          (P')(max[S, restrict[[T, T], [S, S], boolean](≤)](P'))
{2}   empty?[S](P')
```

Expanding the definition of extend,
which is trivially true.
This completes the proof of `max_stable_under_extension.1`.
`max_stable_under_extension.2:`

```
{-1}  is_finite[S](P')
{1}   ∀ (x: (extend[T, S, bool, FALSE](P'))):
        x ≤ max[S, restrict[[T, T], [S, S], boolean](≤)](P')
{2}   empty?[S](P')
```

Repeatedly Skolemizing and flattening,
Expanding the definition of extend,
Adding type constraints for max[S, restrict[[T, T], [S, S], boolean](<=)](P!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: (x!1),
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `max_stable_under_extension.2`.
Q.E.D.

C.135.3 Min_Max_Extension.min_stable_under_extension

Terse proof for `min_stable_under_extension`.
`min_stable_under_extension:`

```
{1}  ∀ (P: non_empty_finite_set[S]):
       min[T, ≤](extend[T, S, bool, FALSE](P)) =
         min[S, restrict[[T, T], [S, S], boolean](≤)](P)
```

Repeatedly Skolemizing and flattening,
Rewriting using min_lem, matching in *,
Expanding the definition of extend,
Repeatedly Skolemizing and flattening,
Adding type constraints for min[S, restrict[[T, T], [S, S], boolean](<=)](P!1),
Expanding the definition of extend,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating the top quantifier in -3 with the terms: (x!1),
Expanding the definition of restrict,
which is trivially true.
This completes the proof of `min_stable_under_extension`.
Q.E.D.

C.136 Proofs for More_Divides (vfiasco-prelude.pvs)

C.136.1 More_Divides.expt_divides

Terse proof for `expt_divides`.
`expt_divides:`

$$\frac{}{\{1\}\quad \forall\ (m,\ n\colon \text{nat},\ a\colon \text{posnat})\colon\ a > 1 \wedge n \leq m \supset \text{divides}(\text{expt}(a,\ n),\ \text{expt}(a,\ m))}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of divides,
Instantiating the top quantifier in 1 with the terms: expt(a', $m' - n'$),
we get 2 subgoals:
`expt_divides.1:`

$$\frac{\begin{array}{ll}\{-1\} & m' \geq 0\\ \{-2\} & n' \geq 0\\ \{-3\} & a' > 0\\ \{-4\} & a' > 1\\ \{-5\} & n' \leq m'\end{array}}{\{1\}\quad \text{expt}(a',\ m') = \text{expt}(a',\ n') \times \text{expt}(a',\ m' - n')}$$

Rewriting using expt_plus_aux, matching in *,
This completes the proof of `expt_divides.1`.
`expt_divides.2:`

$$\frac{\begin{array}{ll}\{-1\} & m' \geq 0\\ \{-2\} & n' \geq 0\\ \{-3\} & a' > 0\\ \{-4\} & a' > 1\\ \{-5\} & n' \leq m'\end{array}}{\{1\}\quad m' - n' \geq 0}$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `expt_divides.2`.
Q.E.D.

C.136.2 More_Divides.divides_expt_gt

Terse proof for `divides_expt_gt`.
`divides_expt_gt:`

$$\frac{}{\{1\}\quad \forall\ (n,\ i,\ j\colon \text{nat})\colon\ j \leq i \wedge \text{divides}(\text{expt}(2,\ i),\ n) \supset \text{divides}(\text{expt}(2,\ j),\ n)}$$

Repeatedly Skolemizing and flattening,
Applying divides_transitive
Instantiating the top quantifier in -1 with the terms: n', expt(2, i'), expt(2, j'),

Simplifying, rewriting, and recording with decision procedures,
Rewriting using expt_divides, matching in *,
This completes the proof of `divides_expt_gt`.
Q.E.D.

C.137 Proofs for More_Function (vfiasco-prelude.pvs)

C.137.1 More_Function.restrict_to_image_TCC1

Terse proof for `restrict_to_image_TCC1`.
`restrict_to_image_TCC1`:

{1} ∀ (f: [Domain → Range]):
∀ (x_1: Domain): image[Domain, Range](f, fullset[Domain])($f(x_1)$)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `restrict_to_image_TCC1`.
Q.E.D.

C.137.2 More_Function.surjective_restrict_to_image

Terse proof for `surjective_restrict_to_image`.
`surjective_restrict_to_image`:

{1} ∀ (f: [Domain → Range]): surjective?(restrict_to_image(f))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `surjective_restrict_to_image`.
Q.E.D.

C.137.3 More_Function.bijective_restrict_to_image

Terse proof for `bijective_restrict_to_image`.
`bijective_restrict_to_image`:

{1} ∀ (f: [Domain → Range]): injective?(f) ⊃ bijective?(restrict_to_image(f))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `bijective_restrict_to_image`.
Q.E.D.

C.138 Proofs for More_List_Props (vfiasco-prelude.pvs)

C.138.1 More_List_Props.length_is_cons

Terse proof for `length_is_cons`.
`length_is_cons`:

{1} ∀ (l: list[T]): length(l) > 0 ⊃ cons?(l)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_is_cons`.
Q.E.D.

C.138.2 More_List_Props.cons_length

Terse proof for `cons_length`.
`cons_length`:

$$\{1\} \quad \forall\ (l\colon\ (\text{cons?}[T]))\colon\ \text{length}(l)\ >\ 0$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cons_length`.
Q.E.D.

C.138.3 More_List_Props.length_cdr_TCC1

Terse proof for `length_cdr_TCC1`.
`length_cdr_TCC1`:

$$\{1\} \quad \forall\ (l\colon\ \text{list}[T])\colon\ \text{length}(l)\ >\ 0\ \supset\ \text{cons?}[T](l)$$

Repeatedly Skolemizing and flattening,
Rewriting using length_is_cons, matching in *,
This completes the proof of `length_cdr_TCC1`.
Q.E.D.

C.138.4 More_List_Props.length_cdr

Terse proof for `length_cdr`.
`length_cdr`:

$$\{1\} \quad \forall\ (l\colon\ \text{list}[T])\colon\ \text{length}(l)\ >\ 0\ \supset\ \text{length}(\text{cdr}(l))\ =\ \text{length}(l) - 1$$

Repeatedly Skolemizing and flattening,
Expanding the definition of length,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `length_cdr`.
Q.E.D.

C.138.5 More_List_Props.conslist_induction

Terse proof for `conslist_induction`.
`conslist_induction`:

$$\begin{aligned}\{1\} \quad & \forall\ (p\colon\ [(\text{cons?}[T])\ \to\ \text{boolean}])\colon \\ & (\forall\ (t\colon\ T)\colon\ p(\text{cons}(t,\ \text{null}))) \wedge \\ & (\forall\ (t\colon\ T,\ \text{tail}\colon\ (\text{cons?}[T]))\colon\ p(\text{tail})\ \supset\ p(\text{cons}(t,\ \text{tail}))) \\ & \supset\ (\forall\ (l\colon\ (\text{cons?}[T]))\colon\ p(l))\end{aligned}$$

Skolemizing and flattening,
Inducting on l on formula 1,
we get 3 subgoals:
`conslist_induction.1:`

{-1}	$\forall$ (t: T): p'(cons(t, null))
{-2}	$\forall$ (t: T, tail: (cons?$[T]$)): p'(tail) $\supset$ p'(cons(t, tail))
{1}	cons?$[T]$(l')
{2}	p'(l')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `conslist_induction.1`.
`conslist_induction.2:`

{-1}	$\forall$ (t: T): p'(cons(t, null))
{-2}	$\forall$ (t: T, tail: (cons?$[T]$)): p'(tail) $\supset$ p'(cons(t, tail))
{1}	FALSE $\supset$ p'(null)

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `conslist_induction.2`.
`conslist_induction.3:`

{-1}	$\forall$ (t: T): p'(cons(t, null))
{-2}	$\forall$ (t: T, tail: (cons?$[T]$)): p'(tail) $\supset$ p'(cons(t, tail))
{1}	$\forall$ (cons1_var: T, cons2_var: list$[T]$): (cons?$[T]$)(cons2_var) $\supset$ p'(cons2_var)) $\supset$ TRUE $\supset$ p'(cons(cons1_var, cons2_var))

Repeatedly Skolemizing and flattening,
Case splitting on cons2_var!1 = null,
we get 2 subgoals:
`conslist_induction.3.1:`

{-1}	cons2_var$'$ = null
{-2}	cons?$[T]$(cons2_var$'$) $\supset$ p'(cons2_var$'$)
{-3}	$\forall$ (t: T): p'(cons(t, null))
{-4}	$\forall$ (t: T, tail: (cons?$[T]$)): p'(tail) $\supset$ p'(cons(t, tail))
{1}	p'(cons(cons1_var$'$, cons2_var$'$))

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `conslist_induction.3.1`.
`conslist_induction.3.2:`

{-1}	cons?$[T]$(cons2_var$'$) $\supset$ p'(cons2_var$'$)
{-2}	$\forall$ (t: T): p'(cons(t, null))
{-3}	$\forall$ (t: T, tail: (cons?$[T]$)): p'(tail) $\supset$ p'(cons(t, tail))
{1}	cons2_var$'$ = null
{2}	p'(cons(cons1_var$'$, cons2_var$'$))

Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `conslist_induction.3.2`.
Q.E.D.

C.138.6 More_List_Props.head_TCC1

Terse proof for `head_TCC1`.
`head_TCC1:`

$$\{1\}\quad \forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{upto}(\mathrm{length}[T](l)))\colon\ \neg\ n\ =\ 0\ \supset\ \mathrm{cons?}[T](l)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `head_TCC1`.
Q.E.D.

C.138.7 More_List_Props.head_TCC2

Terse proof for `head_TCC2`.
`head_TCC2:`

$$\{1\}\quad \forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{upto}(\mathrm{length}[T](l)))\colon \\ \neg\ n\ =\ 0\ \supset\ n-1\ \geq\ 0\ \wedge\ n-1\ \leq\ \mathrm{length}[T](\mathrm{cdr}[T](l))$$

Repeatedly Skolemizing and flattening,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `head_TCC2`.
Q.E.D.

C.138.8 More_List_Props.head_TCC3

Terse proof for `head_TCC3`.
`head_TCC3:`

$$\{1\}\quad \forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{upto}(\mathrm{length}[T](l)))\colon\ \neg\ n\ =\ 0\ \supset\ n-1\ <\ n$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `head_TCC3`.
Q.E.D.

C.138.9 More_List_Props.head_tail

Terse proof for `head_tail`.
`head_tail:`

$$\{1\}\quad \forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{nat})\colon\ n\ \leq\ \mathrm{length}(l)\ \supset\ \mathrm{append}(\mathrm{head}(l,\ n),\ \mathrm{tail}(l,\ n))\ =\ l$$

Inducting on l on formula 1,
we get 2 subgoals:
`head_tail.1:`

$$\{1\}\quad \forall\ (n\colon\ \mathrm{nat})\colon\ n\ \leq\ \mathrm{length}(\mathrm{null})\ \supset\ \mathrm{append}(\mathrm{head}(\mathrm{null},\ n),\ \mathrm{tail}(\mathrm{null},\ n))\ =\ \mathrm{null}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `head_tail.1`.

`head_tail.2:`

```
{1}  ∀ (cons1_var: T, cons2_var: list⌊T⌋):
       (∀ (n: nat):
            n ≤ length(cons2_var) ⊃
             append(head(cons2_var, n), tail(cons2_var, n)) = cons2_var)
        ⊃
        (∀ (n: nat):
             n ≤ length(cons(cons1_var, cons2_var)) ⊃
              append(head(cons(cons1_var, cons2_var), n),
                     tail(cons(cons1_var, cons2_var), n))
               = cons(cons1_var, cons2_var))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of length,
Expanding the definition of head,
Expanding the definition of tail,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`head_tail.2.1:`

```
{-1}  n′ ≥ 0
{-2}  ∀ (n: nat):
        n ≤ length(cons2_var′) ⊃
         append(head(cons2_var′, n), tail(cons2_var′, n)) = cons2_var′
{-3}  n′ ≤ 1 + length(cons2_var′)
{-4}  n′ = 0
-------------------------------------------------------------------------
 {1}  append(null, cons(cons1_var′, cons2_var′)) = cons(cons1_var′, cons2_var′)
```

Expanding the definition of append,
which is trivially true.
This completes the proof of `head_tail.2.1`.
`head_tail.2.2:`

```
{-1}  n′ ≥ 0
{-2}  ∀ (n: nat):
        n ≤ length(cons2_var′) ⊃
         append(head(cons2_var′, n), tail(cons2_var′, n)) = cons2_var′
{-3}  n′ ≤ 1 + length(cons2_var′)
-------------------------------------------------------------------------
 {1}  n′ = 0
 {2}  append(cons(cons1_var′, head(cons2_var′, n′ − 1)),
             tail(cons2_var′, n′ − 1))
       = cons(cons1_var′, cons2_var′)
```

Expanding the definition of append,
Rewriting using -2, matching in *,
This completes the proof of `head_tail.2.2`.
Q.E.D.

C.138.10 More_List_Props.length_head

Terse proof for `length_head`.

`length_head:`

$$\{1\} \quad \forall\ (l\colon \text{list}[T],\ n\colon \text{nat})\colon\ n \leq \text{length}(l) \supset \text{length}(\text{head}(l,\ n)) = n$$

Inducting on l on formula 1,
we get 2 subgoals:
`length_head.1:`

$$\{1\} \quad \forall\ (n\colon \text{nat})\colon\ n \leq \text{length}(\text{null}) \supset \text{length}(\text{head}(\text{null},\ n)) = n$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_head.1`.
`length_head.2:`

$$\begin{array}{ll}\{1\} & \forall\ (\text{cons1\_var}\colon T,\ \text{cons2\_var}\colon \text{list}[T])\colon \\ & \quad (\forall\ (n\colon \text{nat})\colon\ n \leq \text{length}(\text{cons2\_var}) \supset \text{length}(\text{head}(\text{cons2\_var},\ n)) = n) \supset \\ & \quad\ (\forall\ (n\colon \text{nat})\colon \\ & \qquad n \leq \text{length}(\text{cons}(\text{cons1\_var},\ \text{cons2\_var})) \supset \\ & \qquad\ \text{length}(\text{head}(\text{cons}(\text{cons1\_var},\ \text{cons2\_var}),\ n)) = n)\end{array}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of head,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -2 with the terms: (n!1 - 1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_head.2`.
Q.E.D.

C.138.11 More_List_Props.length_tail

Terse proof for `length_tail`.
`length_tail:`

$$\begin{array}{ll}\{1\} & \forall\ (l\colon \text{list}[T],\ n\colon \text{nat})\colon \\ & \quad n \leq \text{length}(l) \supset \text{length}(\text{tail}(l,\ n)) = \text{length}(l) - n\end{array}$$

Inducting on l on formula 1,
we get 2 subgoals:
`length_tail.1:`

$$\{1\} \quad \forall\ (n\colon \text{nat})\colon\ n \leq \text{length}(\text{null}) \supset \text{length}(\text{tail}(\text{null},\ n)) = \text{length}(\text{null}) - n$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_tail.1`.

`length_tail.2:`

{1} ∀ (cons1_var: T, cons2_var: list$[T]$):
(∀ (n: nat):
$n \leq$ length(cons2_var) ⊃
length(tail(cons2_var, n)) = length(cons2_var) − n)
⊃
(∀ (n: nat):
$n \leq$ length(cons(cons1_var, cons2_var)) ⊃
length(tail(cons(cons1_var, cons2_var), n)) =
length(cons(cons1_var, cons2_var)) − n)

Repeatedly Skolemizing and flattening,
Expanding the definition of tail,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`length_tail.2.1:`

{-1} $n' \geq 0$
{-2} ∀ (n: nat):
$n \leq$ length(cons2_var′) ⊃
length(tail(cons2_var′, n)) = length(cons2_var′) − n
{-3} $n' \leq$ length(cons(cons1_var′, cons2_var′))
{-4} null?(tail(cons2_var′, $n' - 1$))

{1} $n' = 0$
{2} $0 = 1 +$ length(cons2_var′) − n'

Instantiating the top quantifier in -2 with the terms: (n!1 - 1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_tail.2.1`.
`length_tail.2.2:`

{-1} $n' \geq 0$
{-2} ∀ (n: nat):
$n \leq$ length(cons2_var′) ⊃
length(tail(cons2_var′, n)) = length(cons2_var′) − n
{-3} $n' \leq$ length(cons(cons1_var′, cons2_var′))

{1} $n' = 0$
{2} null?(tail(cons2_var′, $n' - 1$))
{3} length(cdr(tail(cons2_var′, $n' - 1$))) = length(cons2_var′) − n'

Instantiating the top quantifier in -2 with the terms: (n!1 - 1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `length_tail.2.2`.
Q.E.D.

C.138.12 More_List_Props.nth_head_TCC1

Terse proof for `nth_head_TCC1`.
`nth_head_TCC1:`

{1} ∀ (l: list$[T]$, n, i: nat):
$n \leq$ length(l) ∧ $i < n$ ⊃ $i <$ length$[T]$(head(l, n))

Repeatedly Skolemizing and flattening,
Rewriting using length_head, matching in *,
This completes the proof of `nth_head_TCC1`.
Q.E.D.

C.138.13 More_List_Props.nth_head_TCC2

Terse proof for `nth_head_TCC2`.
`nth_head_TCC2:`

$$\{1\}\quad \forall\ (l\colon \mathrm{list}[T],\ n,\ i\colon \mathrm{nat})\colon\ n \le \mathrm{length}(l) \wedge i < n \supset i < \mathrm{length}[T](l)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_head_TCC2`.
Q.E.D.

C.138.14 More_List_Props.nth_head

Terse proof for `nth_head`.
`nth_head:`

$$\{1\}\quad \forall\ (l\colon \mathrm{list}[T],\ n,\ i\colon \mathrm{nat})\colon\ n \le \mathrm{length}(l) \wedge i < n \supset \mathrm{nth}(\mathrm{head}(l,\ n),\ i) = \mathrm{nth}(l,\ i)$$

Inducting on n on formula 1,
we get 4 subgoals:
`nth_head.1:`

$$\{1\}\quad \forall\ (l\colon \mathrm{list}[T],\ i\colon \mathrm{nat})\colon\ 0 \le \mathrm{length}(l) \wedge i < 0 \supset \mathrm{nth}(\mathrm{head}(l,\ 0),\ i) = \mathrm{nth}(l,\ i)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_head.1`.
`nth_head.2:`

$$\begin{array}{l}\{1\}\quad \forall\ j\colon\\ \quad (\forall\ (l\colon \mathrm{list}[T],\ i\colon \mathrm{nat})\colon\\ \qquad j \le \mathrm{length}(l) \wedge i < j \supset \mathrm{nth}(\mathrm{head}(l,\ j),\ i) = \mathrm{nth}(l,\ i))\\ \quad \supset\\ \quad (\forall\ (l\colon \mathrm{list}[T],\ i\colon \mathrm{nat})\colon\\ \qquad j+1 \le \mathrm{length}(l) \wedge i < j+1 \supset\\ \qquad \mathrm{nth}(\mathrm{head}(l,\ j+1),\ i) = \mathrm{nth}(l,\ i))\end{array}$$

Repeatedly Skolemizing and flattening,
Expanding the definition of head,
Expanding the definition of nth,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using -3, matching in *,
Rewriting using length_cdr, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `nth_head.2`.

`nth_head.3`:

$$\begin{array}{ll}\{1\} & \forall\ (n\colon \text{nat},\ l\colon \text{list}[T],\ i\colon \text{nat})\colon\ n \le \text{length}(l) \wedge i < n \supset i < \text{length}[T](l)\\ \{2\} & \forall\ (l\colon \text{list}[T],\ i\colon \text{nat})\colon\\ & \quad n' \le \text{length}(l) \wedge i < n' \supset \text{nth}(\text{head}(l,\ n'),\ i) = \text{nth}(l,\ i)\end{array}$$

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_head.3`.
`nth_head.4`:

$$\begin{array}{ll}\{1\} & \forall\ (n\colon \text{nat},\ l\colon \text{list}[T],\ i\colon \text{nat})\colon\\ & \quad n \le \text{length}(l) \wedge i < n \supset i < \text{length}[T](\text{head}(l,\ n))\\ \{2\} & \forall\ (l\colon \text{list}[T],\ i\colon \text{nat})\colon\\ & \quad n' \le \text{length}(l) \wedge i < n' \supset \text{nth}(\text{head}(l,\ n'),\ i) = \text{nth}(l,\ i)\end{array}$$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using length_head, matching in *,
This completes the proof of `nth_head.4`.
Q.E.D.

C.138.15 More_List_Props.nth_tail_TCC1

Terse proof for `nth_tail_TCC1`.
`nth_tail_TCC1`:

$$\begin{array}{ll}\{1\} & \forall\ (l\colon \text{list}[T],\ n,\ i\colon \text{nat})\colon\\ & \quad n \le \text{length}(l) \wedge i + n < \text{length}(l) \supset i < \text{length}[T](\text{tail}(l,\ n))\end{array}$$

Repeatedly Skolemizing and flattening,
Rewriting using length_tail, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_tail_TCC1`.
Q.E.D.

C.138.16 More_List_Props.nth_tail_TCC2

Terse proof for `nth_tail_TCC2`.
`nth_tail_TCC2`:

$$\begin{array}{ll}\{1\} & \forall\ (l\colon \text{list}[T],\ n,\ i\colon \text{nat})\colon\\ & \quad n \le \text{length}(l) \wedge i + n < \text{length}(l) \supset n + i < \text{length}[T](l)\end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_tail_TCC2`.
Q.E.D.

C.138.17 More_List_Props.nth_tail

Terse proof for `nth_tail`.

`nth_tail:`

{1} ∀ (l: list$[T]$, n, i: nat):
$n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset$
$\text{nth}(\text{tail}(l, n), i) = \text{nth}(l, n + i)$

Inducting on n on formula 1,
we get 4 subgoals:
`nth_tail.1:`

{1} ∀ (l: list$[T]$, i: nat):
$0 \leq \text{length}(l) \wedge i + 0 < \text{length}(l) \supset$
$\text{nth}(\text{tail}(l, 0), i) = \text{nth}(l, 0 + i)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_tail.1`.
`nth_tail.2:`

{1} ∀ j:
(∀ (l: list$[T]$, i: nat):
$j \leq \text{length}(l) \wedge i + j < \text{length}(l) \supset$
$\text{nth}(\text{tail}(l, j), i) = \text{nth}(l, j + i)$)
⊃
(∀ (l: list$[T]$, i: nat):
$j + 1 \leq \text{length}(l) \wedge i + j + 1 < \text{length}(l) \supset$
$\text{nth}(\text{tail}(l, j + 1), i) = \text{nth}(l, j + 1 + i)$)

Repeatedly Skolemizing and flattening,
Expanding the definition of tail,
Expanding the definition of nth,
Installing automatic rewrites from: -3 length_cdr
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `nth_tail.2`.
`nth_tail.3:`

{1} ∀ (n: nat, l: list$[T]$, i: nat):
$n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset i + n < \text{length}[T](l)$
{2} ∀ (l: list$[T]$, i: nat):
$n' \leq \text{length}(l) \wedge i + n' < \text{length}(l) \supset$
$\text{nth}(\text{tail}(l, n'), i) = \text{nth}(l, n' + i)$

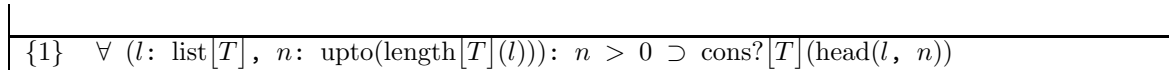
Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_tail.3`.
`nth_tail.4:`

{1} ∀ (n: nat, l: list$[T]$, i: nat):
$n \leq \text{length}(l) \wedge i + n < \text{length}(l) \supset i < \text{length}[T](\text{tail}(l, n))$
{2} ∀ (l: list$[T]$, i: nat):
$n' \leq \text{length}(l) \wedge i + n' < \text{length}(l) \supset$
$\text{nth}(\text{tail}(l, n'), i) = \text{nth}(l, n' + i)$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using length_tail, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_tail.4`.
Q.E.D.

C.138.18 More_List_Props.car_head_TCC1

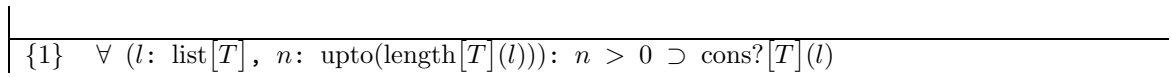
Terse proof for `car_head_TCC1`.
`car_head_TCC1:`



Repeatedly Skolemizing and flattening,
Expanding the definition of head,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `car_head_TCC1`.
Q.E.D.

C.138.19 More_List_Props.car_head_TCC2

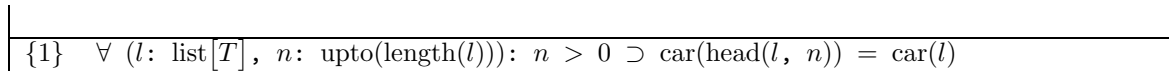
Terse proof for `car_head_TCC2`.
`car_head_TCC2:`



Repeatedly Skolemizing and flattening,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `car_head_TCC2`.
Q.E.D.

C.138.20 More_List_Props.car_head

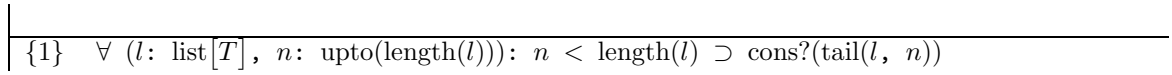
Terse proof for `car_head`.
`car_head:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `car_head`.
Q.E.D.

C.138.21 More_List_Props.cons_tail

Terse proof for `cons_tail`.
`cons_tail:`



Repeatedly Skolemizing and flattening,
Using lemma length_tail,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cons_tail`.
Q.E.D.

C.138.22 More_List_Props.car_tail_TCC1

Terse proof for `car_tail_TCC1`.
`car_tail_TCC1:`

```
{1}  ∀ (l: list[T], n: upto(length[T](l))):
       n < length(l) ⊃ cons?[T](tail(l, n))
```

Repeatedly Skolemizing and flattening,
Using lemma length_tail,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `car_tail_TCC1`.
Q.E.D.

C.138.23 More_List_Props.car_tail

Terse proof for `car_tail`.
`car_tail:`

```
{1}  ∀ (l: list[T], n: upto(length(l))):
       n < length(l) ⊃ car(tail(l, n)) = nth(l, n)
```

Inducting on l on formula 1,
we get 3 subgoals:
`car_tail.1:`

```
{1}  ∀ (n: upto(length(null))):
       n < length(null) ⊃ car(tail(null, n)) = nth(null, n)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `car_tail.1`.
`car_tail.2:`

```
{1}  ∀ (cons1_var: T, cons2_var: list[T]):
       (∀ (n: upto(length(cons2_var))):
          n < length(cons2_var) ⊃ car(tail(cons2_var, n)) = nth(cons2_var, n))
       ⊃
       (∀ (n: upto(length(cons(cons1_var, cons2_var)))):
          n < length(cons(cons1_var, cons2_var)) ⊃
            car(tail(cons(cons1_var, cons2_var), n)) = nth(cons(cons1_var, cons2_var), n))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of tail,
Expanding the definition of nth,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using -2, matching in *,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `car_tail.2`.
`car_tail.3`:

{1} $\forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{upto}(\mathrm{length}[T](l)))\colon$
$n\ <\ \mathrm{length}(l)\ \supset\ \mathrm{cons?}[T](\mathrm{tail}(l,\ n))$
{2} $\forall\ (n\colon\ \mathrm{upto}(\mathrm{length}(l'))):\ n\ <\ \mathrm{length}(l')\ \supset\ \mathrm{car}(\mathrm{tail}(l',\ n))\ =\ \mathrm{nth}(l',\ n)$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Using lemma length_tail,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `car_tail.3`.
Q.E.D.

C.138.24 More_List_Props.cdr_tail_TCC1

Terse proof for `cdr_tail_TCC1`.
`cdr_tail_TCC1`:

{1} $\forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{upto}(\mathrm{length}[T](l)))\colon$
$n\ <\ \mathrm{length}(l)\ \supset\ n+1\ \leq\ \mathrm{length}[T](l)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cdr_tail_TCC1`.
Q.E.D.

C.138.25 More_List_Props.cdr_tail

Terse proof for `cdr_tail`.
`cdr_tail`:

{1} $\forall\ (l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{upto}(\mathrm{length}(l)))\colon$
$n\ <\ \mathrm{length}(l)\ \supset\ \mathrm{cdr}(\mathrm{tail}(l,\ n))\ =\ \mathrm{tail}(l,\ n+1)$

Inducting on l on formula 1,
we get 4 subgoals:
`cdr_tail.1`:

{1} $\forall\ (n\colon\ \mathrm{upto}(\mathrm{length}(\mathrm{null})))\colon$
$n\ <\ \mathrm{length}(\mathrm{null})\ \supset\ \mathrm{cdr}(\mathrm{tail}(\mathrm{null},\ n))\ =\ \mathrm{tail}(\mathrm{null},\ n+1)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `cdr_tail.1`.

`cdr_tail.2:`

```
{1}  ∀ (cons1_var: T, cons2_var: list[T]):
       (∀ (n: upto(length(cons2_var))):
          n < length(cons2_var) ⊃
           cdr(tail(cons2_var, n)) = tail(cons2_var, n + 1))
        ⊃
        (∀ (n: upto(length(cons(cons1_var, cons2_var)))):
           n < length(cons(cons1_var, cons2_var)) ⊃
            cdr(tail(cons(cons1_var, cons2_var), n)) =
             tail(cons(cons1_var, cons2_var), n + 1))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of tail,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`cdr_tail.2.1:`

```
{-1}  n′ ≤ length(cons(cons1_var′, cons2_var′))
{-2}  ∀ (n: upto(length(cons2_var′))):
        n < length(cons2_var′) ⊃
         cdr(tail(cons2_var′, n)) = tail(cons2_var′, 1 + n)
{-3}  n′ < length(cons(cons1_var′, cons2_var′))
{-4}  n′ = 0
------------------------------------------------------------
 {1}  cons2_var′ = tail(cons2_var′, n′)
```

Expanding the definition of tail,
which is trivially true.
This completes the proof of `cdr_tail.2.1`.
`cdr_tail.2.2:`

```
{-1}  n′ ≤ length(cons(cons1_var′, cons2_var′))
{-2}  ∀ (n: upto(length(cons2_var′))):
        n < length(cons2_var′) ⊃
         cdr(tail(cons2_var′, n)) = tail(cons2_var′, 1 + n)
{-3}  n′ < length(cons(cons1_var′, cons2_var′))
------------------------------------------------------------
 {1}  n′ = 0
 {2}  cdr(tail(cons2_var′, n′ − 1)) = tail(cons2_var′, n′)
```

Rewriting using -2, matching in *,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cdr_tail.2.2`.
`cdr_tail.3:`

```
{1}  ∀ (l: list[T], n: upto(length[T](l))):
       n < length(l) ⊃ 1 + n ≤ length[T](l)
{2}  ∀ (n: upto(length(l′))):
       n < length(l′) ⊃ cdr(tail(l′, n)) = tail(l′, n + 1)
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `cdr_tail.3`.

`cdr_tail.4`:

{1} ∀ (l: list[T], n: upto(length[T](l))):
n < length(l) ⊃ cons?[T](tail(l, n))
{2} ∀ (n: upto(length(l'))):
n < length(l') ⊃ cdr(tail(l', n)) = tail(l', $n+1$)

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Rewriting using cons_tail, matching in *,
This completes the proof of `cdr_tail.4`.
Q.E.D.

C.138.26 More_List_Props.every_implied

Terse proof for `every_implied`.

`every_implied`:

{1} ∀ (l: list[T], P, Q: PRED[T]):
(∀ (t: T): $P(t)$ ⊃ $Q(t)$) ∧ every(P)(l) ⊃ every(Q)(l)

Inducting on l on formula 1,
we get 2 subgoals:

`every_implied.1`:

{1} ∀ (P, Q: PRED[T]):
(∀ (t: T): $P(t)$ ⊃ $Q(t)$) ∧ every(P)(null) ⊃ every(Q)(null)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `every_implied.1`.

`every_implied.2`:

{1} ∀ (cons1_var: T, cons2_var: list[T]):
(∀ (P, Q: PRED[T]):
(∀ (t: T): $P(t)$ ⊃ $Q(t)$) ∧ every(P)(cons2_var) ⊃
every(Q)(cons2_var))
⊃
(∀ (P, Q: PRED[T]):
(∀ (t: T): $P(t)$ ⊃ $Q(t)$) ∧ every(P)(cons(cons1_var, cons2_var)) ⊃
every(Q)(cons(cons1_var, cons2_var)))

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Instantiating the top quantifier in -1 with the terms: (P!1 Q!1),
Replacing using formula -2,
Instantiating quantified variables,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `every_implied.2`.
Q.E.D.

C.138.27 More_List_Props.every_conjunct_left

Terse proof for `every_conjunct_left`.

`every_conjunct_left:`

{1} $\forall\ (l\colon \text{list}[T],\ P,\ Q\colon \text{PRED}[T])\colon$
$\text{every}(P)(l)\ \wedge\ \text{every}(Q)(l)\ \supset\ \text{every}(\lambda\ (t\colon T)\colon\ P(t)\ \wedge\ Q(t))(l)$

Inducting on l on formula 1,

we get 2 subgoals:

`every_conjunct_left.1:`

{1} $\forall\ (P,\ Q\colon \text{PRED}[T])\colon$
$\text{every}(P)(\text{null})\ \wedge\ \text{every}(Q)(\text{null})\ \supset\ \text{every}(\lambda\ (t\colon T)\colon\ P(t)\ \wedge\ Q(t))(\text{null})$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_conjunct_left.1`.

`every_conjunct_left.2:`

{1} $\forall\ (\text{cons1\_var}\colon T,\ \text{cons2\_var}\colon \text{list}[T])\colon$
$(\forall\ (P,\ Q\colon \text{PRED}[T])\colon$
$\text{every}(P)(\text{cons2\_var})\ \wedge\ \text{every}(Q)(\text{cons2\_var})\ \supset$
$\text{every}(\lambda\ (t\colon T)\colon\ P(t)\ \wedge\ Q(t))(\text{cons2\_var}))$
$\supset$
$(\forall\ (P,\ Q\colon \text{PRED}[T])\colon$
$\text{every}(P)(\text{cons}(\text{cons1\_var},\ \text{cons2\_var}))\ \wedge\ \text{every}(Q)(\text{cons}(\text{cons1\_var},\ \text{cons2\_var}))$
$\supset\ \text{every}(\lambda\ (t\colon T)\colon\ P(t)\ \wedge\ Q(t))(\text{cons}(\text{cons1\_var},\ \text{cons2\_var})))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_conjunct_left.2`.

Q.E.D.

C.138.28 More_List_Props.every_iff_forall

Terse proof for `every_iff_forall`.

`every_iff_forall:`

{1} $\forall\ (l\colon \text{list}[T],\ P\colon \text{PRED}[T])\colon$
$\text{every}(P)(l)\ \equiv\ (\forall\ (t\colon T)\colon\ \text{member}(t,\ l)\ \supset\ P(t))$

Inducting on l on formula 1,

we get 2 subgoals:

`every_iff_forall.1:`

{1} $\forall\ (P\colon \text{PRED}[T])\colon\ \text{every}(P)(\text{null})\ \equiv\ (\forall\ (t\colon T)\colon\ \text{member}(t,\ \text{null})\ \supset\ P(t))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `every_iff_forall.1`.

`every_iff_forall.2:`

```
{1}  ∀ (cons1_var: T, cons2_var: list[T]):
       (∀ (P: PRED[T]):
            every(P)(cons2_var) ≡ (∀ (t: T): member(t, cons2_var) ⊃ P(t)))
        ⊃
       (∀ (P: PRED[T]):
            every(P)(cons(cons1_var, cons2_var)) ≡
              (∀ (t: T): member(t, cons(cons1_var, cons2_var)) ⊃ P(t)))
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of every,
Expanding the definition of member,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`every_iff_forall.2.1:`

```
{-1}  every(P′)(cons2_var′)
{-2}  ∀ (t: T): member(t, cons2_var′) ⊃ P′(t)
{-3}  P′(cons1_var′)
{1}   ∀ (t: T): t = cons1_var′ ∨ member(t, cons2_var′) ⊃ P′(t)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `every_iff_forall.2.1`.
`every_iff_forall.2.2:`

```
{-1}  every(P′)(cons2_var′)
{-2}  ∀ (t: T): member(t, cons2_var′) ⊃ P′(t)
{-3}  ∀ (t: T): t = cons1_var′ ∨ member(t, cons2_var′) ⊃ P′(t)
{1}   P′(cons1_var′)
```

Instantiating quantified variables,
This completes the proof of `every_iff_forall.2.2`.
`every_iff_forall.2.3:`

```
{-1}  ∀ (t: T): t = cons1_var′ ∨ member(t, cons2_var′) ⊃ P′(t)
{1}   every(P′)(cons2_var′)
{2}   ∀ (t: T): member(t, cons2_var′) ⊃ P′(t)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `every_iff_forall.2.3`.
Q.E.D.

C.138.29 More_List_Props.some_iff_exists

Terse proof for `some_iff_exists`.

`some_iff_exists:`

{1} $\forall\ (l\colon \text{list}[T],\ P\colon \text{PRED}[T])\colon$
$\text{some}(P)(l) \equiv (\exists\ (t\colon T)\colon \text{member}(t,\ l) \wedge P(t))$

Inducting on l on formula 1,
we get 2 subgoals:
`some_iff_exists.1:`

{1} $\forall\ (P\colon \text{PRED}[T])\colon \text{some}(P)(\text{null}) \equiv (\exists\ (t\colon T)\colon \text{member}(t,\ \text{null}) \wedge P(t))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `some_iff_exists.1`.
`some_iff_exists.2:`

{1} $\forall\ (\text{cons1\_var}\colon T,\ \text{cons2\_var}\colon \text{list}[T])\colon$
$(\forall\ (P\colon \text{PRED}[T])\colon$
$\text{some}(P)(\text{cons2\_var}) \equiv (\exists\ (t\colon T)\colon \text{member}(t,\ \text{cons2\_var}) \wedge P(t)))$
$\supset$
$(\forall\ (P\colon \text{PRED}[T])\colon$
$\text{some}(P)(\text{cons}(\text{cons1\_var},\ \text{cons2\_var})) \equiv$
$(\exists\ (t\colon T)\colon \text{member}(t,\ \text{cons}(\text{cons1\_var},\ \text{cons2\_var})) \wedge P(t)))$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Expanding the definition of some,
Expanding the definition of member,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`some_iff_exists.2.1:`

{-1} $\text{some}(P')(\text{cons2\_var}')$
{-2} $\exists\ (t\colon T)\colon \text{member}(t,\ \text{cons2\_var}') \wedge P'(t)$
{1} $\exists\ (t\colon T)\colon (t = \text{cons1\_var}' \vee \text{member}(t,\ \text{cons2\_var}')) \wedge P'(t)$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (t!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `some_iff_exists.2.1`.
`some_iff_exists.2.2:`

{-1} $P'(\text{cons1\_var}')$
{1} $\text{some}(P')(\text{cons2\_var}')$
{2} $\exists\ (t\colon T)\colon \text{member}(t,\ \text{cons2\_var}') \wedge P'(t)$
{3} $\exists\ (t\colon T)\colon (t = \text{cons1\_var}' \vee \text{member}(t,\ \text{cons2\_var}')) \wedge P'(t)$

Instantiating the top quantifier in 3 with the terms: (cons1_var!1),
which is trivially true.
This completes the proof of `some_iff_exists.2.2`.
`some_iff_exists.2.3:`

{-1} $\exists\ (t\colon T)\colon (t = \text{cons1\_var}' \vee \text{member}(t,\ \text{cons2\_var}')) \wedge P'(t)$
{1} $\text{some}(P')(\text{cons2\_var}')$
{2} $\exists\ (t\colon T)\colon \text{member}(t,\ \text{cons2\_var}') \wedge P'(t)$
{3} $P'(\text{cons1\_var}')$

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `some_iff_exists.2.3`.

Q.E.D.

C.138.30 More_List_Props.list_extensionality_TCC1

Terse proof for `list_extensionality_TCC1`.

`list_extensionality_TCC1:`

{1} $\forall\ (l_1,\ l_2\colon \text{list}[T])\colon$
$\text{length}(l_1) = \text{length}(l_2) \supset$
$(\forall\ (i\colon \text{below}(\text{length}[T](l_1)))\colon\ i < \text{length}[T](l_2))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `list_extensionality_TCC1`.

Q.E.D.

C.138.31 More_List_Props.list_extensionality

Terse proof for `list_extensionality`.

`list_extensionality:`

{1} $\forall\ (l_1,\ l_2\colon \text{list}[T])\colon$
$l_1 = l_2 \equiv$
$(\text{length}(l_1) = \text{length}(l_2) \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(l_1)))\colon\ \text{nth}(l_1,\ i) = \text{nth}(l_2,\ i)))$

Inducting on l_1 on formula 1,

we get 3 subgoals:

`list_extensionality.1:`

{1} $\forall\ (l_2\colon \text{list}[T])\colon$
$\text{null} = l_2 \equiv$
$(\text{length}(\text{null}) = \text{length}(l_2) \wedge$
$(\forall\ (i\colon \text{below}(\text{length}(\text{null})))\colon\ \text{nth}(\text{null},\ i) = \text{nth}(l_2,\ i)))$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `list_extensionality.1`.

`list_extensionality.2:`

{1} ∀ (cons1_var: T, cons2_var: list$[T]$):
(∀ (l_2: list$[T]$):
cons2_var = l_2 ≡
(length(cons2_var) = length(l_2) ∧
(∀ (i: below(length(cons2_var))): nth(cons2_var, i) = nth(l_2, i))))
⊃
(∀ (l_2: list$[T]$):
cons(cons1_var, cons2_var) = l_2 ≡
(length(cons(cons1_var, cons2_var)) = length(l_2) ∧
(∀ (i: below(length(cons(cons1_var, cons2_var)))):
nth(cons(cons1_var, cons2_var), i) = nth(l_2, i))))

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of length,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -1 with the terms: cdr(l_2'),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`list_extensionality.2.1:`

{-1} cons2_var′ = cdr(l_2')
{-2} length(cons2_var′) = length(cdr(l_2'))
{-3} ∀ (i: below(length(cons2_var′))): nth(cons2_var′, i) = nth(cdr(l_2'), i)
{-4} ∀ (i: below(length(cons(cons1_var′, cons2_var′)))):
nth(cons(cons1_var′, cons2_var′), i) = nth(l_2', i)

{1} cons(cons1_var′, cons2_var′) = l_2'
{2} null?(l_2')

Applying extensionality,
Instantiating the top quantifier in -4 with the terms: 0,
we get 2 subgoals:
`list_extensionality.2.1.1:`

{-1} cons2_var′ = cdr(l_2')
{-2} length(cons2_var′) = length(cdr(l_2'))
{-3} ∀ (i: below(length(cons2_var′))): nth(cons2_var′, i) = nth(cdr(l_2'), i)
{-4} nth(cons(cons1_var′, cons2_var′), 0) = nth(l_2', 0)

{1} cons1_var′ = car(l_2')
{2} null?(l_2')

Expanding the definition of nth,
which is trivially true.
This completes the proof of `list_extensionality.2.1.1`.
`list_extensionality.2.1.2:`

{-1} cons2_var′ = cdr(l_2')
{-2} length(cons2_var′) = length(cdr(l_2'))
{-3} ∀ (i: below(length(cons2_var′))): nth(cons2_var′, i) = nth(cdr(l_2'), i)

{1} 0 < length$[T]$(cons$[T]$(cons1_var′, cons2_var′))
{2} cons1_var′ = car(l_2')
{3} null?(l_2')

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `list_extensionality.2.1.2`.
`list_extensionality.2.2:`

```
{-1}  length(cons2_var') = length(cdr(l'_2))
{-2}  ∀ (i: below(length(cons(cons1_var', cons2_var')))):
        nth(cons(cons1_var', cons2_var'), i) = nth(l'_2, i)
------------------------------------------------------------
{1}   cons2_var' = cdr(l'_2)
{2}   ∀ (i: below(length(cons2_var'))): nth(cons2_var', i) = nth(cdr(l'_2), i)
{3}   cons(cons1_var', cons2_var') = l'_2
{4}   null?(l'_2)
```

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -3 with the terms: $i' + 1$,
we get 2 subgoals:
`list_extensionality.2.2.1:`

```
{-1}  i' < length(cons2_var')
{-2}  length(cons2_var') = length(cdr(l'_2))
{-3}  nth(cons(cons1_var', cons2_var'), i' + 1) = nth(l'_2, i' + 1)
------------------------------------------------------------
{1}   cons2_var' = cdr(l'_2)
{2}   nth(cons2_var', i') = nth(cdr(l'_2), i')
{3}   cons(cons1_var', cons2_var') = l'_2
{4}   null?(l'_2)
```

Expanding the definition of nth,
which is trivially true.
This completes the proof of `list_extensionality.2.2.1`.
`list_extensionality.2.2.2:`

```
{-1}  i' < length(cons2_var')
{-2}  length(cons2_var') = length(cdr(l'_2))
------------------------------------------------------------
{1}   1 + i' < length[T](cons[T](cons1_var', cons2_var'))
{2}   cons2_var' = cdr(l'_2)
{3}   nth(cons2_var', i') = nth(cdr(l'_2), i')
{4}   cons(cons1_var', cons2_var') = l'_2
{5}   null?(l'_2)
```

Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `list_extensionality.2.2.2`.
`list_extensionality.3:`

```
------------------------------------------------------------
{1}   ∀ (l_1, l_2: list[T]):
        length(l_1) = length(l_2) ⊃
          (∀ (i: below(length[T](l_1))): i < length[T](l_2))
{2}   ∀ (l_2: list[T]):
        l'_1 = l_2 ≡
          (length(l'_1) = length(l_2) ∧
             (∀ (i: below(length(l'_1))): nth(l'_1, i) = nth(l_2, i)))
```

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `list_extensionality.3`.
Q.E.D.

C.138.32 More_List_Props.list_remove_TCC1

Terse proof for `list_remove_TCC1`.
`list_remove_TCC1`:

{1} $\forall\ (x\colon T,\ l\colon \mathrm{list}[T],\ \mathrm{car}\colon T,\ \mathrm{cdr}\colon \mathrm{list}[T])\colon$
$x = \mathrm{car} \wedge l = \mathrm{cons}(\mathrm{car},\ \mathrm{cdr}) \supset \mathrm{length}[T](\mathrm{cdr}) < \mathrm{length}[T](l)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `list_remove_TCC1`.
Q.E.D.

C.138.33 More_List_Props.list_remove_TCC2

Terse proof for `list_remove_TCC2`.
`list_remove_TCC2`:

{1} $\forall\ (x\colon T,\ l\colon \mathrm{list}[T],\ \mathrm{car}\colon T,\ \mathrm{cdr}\colon \mathrm{list}[T])\colon$
$\neg\ x = \mathrm{car} \wedge l = \mathrm{cons}(\mathrm{car},\ \mathrm{cdr}) \supset \mathrm{length}[T](\mathrm{cdr}) < \mathrm{length}[T](l)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `list_remove_TCC2`.
Q.E.D.

C.138.34 More_List_Props.list_remove_commutative

Terse proof for `list_remove_commutative`.
`list_remove_commutative`:

{1} $\forall\ (x,\ y\colon T,\ l\colon \mathrm{list}[T])\colon$
$\mathrm{list\_remove}(x,\ \mathrm{list\_remove}(y,\ l)) = \mathrm{list\_remove}(y,\ \mathrm{list\_remove}(x,\ l))$

By induction on l, and by repeatedly rewriting and simplifying,
This completes the proof of `list_remove_commutative`.
Q.E.D.

C.138.35 More_List_Props.member_list_remove

Terse proof for `member_list_remove`.
`member_list_remove`:

{1} $\forall\ (x,\ y\colon T,\ l\colon \mathrm{list}[T])\colon$
$\mathrm{member}(x,\ \mathrm{list\_remove}(y,\ l)) = \textsc{if}\ x = y\ \textsc{then false else}\ \mathrm{member}(x,\ l)\ \textsc{endif}$

By induction on l, and by repeatedly rewriting and simplifying,
This completes the proof of `member_list_remove`.
Q.E.D.

C.138.36 More_List_Props.nth_member

Terse proof for `nth_member`.

`nth_member:`

```
{1}  ∀ (x: T, l: list[T]):
       member(x, l) ≡ (∃ (i: below(length(l))): nth(l, i) = x)
```

Inducting on l on formula 1,
we get 2 subgoals:

`nth_member.1:`

```
{1}  ∀ (x: T): member(x, null) ≡ (∃ (i: below(length(null))): nth(null, i) = x)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_member.1`.

`nth_member.2:`

```
{1}  ∀ (cons1_var: T, cons2_var: list[T]):
       (∀ (x: T):
          member(x, cons2_var) ≡
           (∃ (i: below(length(cons2_var))): nth(cons2_var, i) = x))
        ⊃
       (∀ (x: T):
          member(x, cons(cons1_var, cons2_var)) ≡
           (∃ (i: below(length(cons(cons1_var, cons2_var)))):
              nth(cons(cons1_var, cons2_var), i) = x))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: member nth length
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:

`nth_member.2.1:`

```
{-1}  ∀ (x: T):
        member(x, cons2_var′) ≡
         (∃ (i: below(length(cons2_var′))): nth(cons2_var′, i) = x)
{-2}  x′ = cons1_var′
------------------------------------------------------------------
 {1}  ∃ (i: below(length(cons(cons1_var′, cons2_var′)))):
        nth(cons(cons1_var′, cons2_var′), i) = x′
```

Instantiating the top quantifier in 1 with the terms: 0,
we get 2 subgoals:

`nth_member.2.1.1:`

```
{-1}  ∀ (x: T):
        member(x, cons2_var′) ≡
         (∃ (i: below(length(cons2_var′))): nth(cons2_var′, i) = x)
{-2}  x′ = cons1_var′
------------------------------------------------------------------
 {1}  nth(cons(cons1_var′, cons2_var′), 0) = x′
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_member.2.1.1`.

`nth_member.2.1.2:`

{-1} ∀ (x: T):
member(x, cons2_var′) ≡
($\exists$ (i: below(length(cons2_var′))): nth(cons2_var′, i) = x)
{-2} x' = cons1_var′

{1} $0 < 1 + \text{length}(\text{cons2\_var}')$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_member.2.1.2`.

`nth_member.2.2:`

{-1} ∀ (x: T):
member(x, cons2_var′) ≡
($\exists$ (i: below(length(cons2_var′))): nth(cons2_var′, i) = x)
{-2} member(x', cons2_var′)

{1} $\exists$ (i: below(length(cons(cons1_var′, cons2_var′)))):
nth(cons(cons1_var′, cons2_var′), i) = x'

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Instantiating the top quantifier in 1 with the terms: $i' + 1$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_member.2.2`.

`nth_member.2.3:`

{-1} ∀ (x: T):
member(x, cons2_var′) ≡
($\exists$ (i: below(length(cons2_var′))): nth(cons2_var′, i) = x)
{-2} $\exists$ (i: below(length(cons(cons1_var′, cons2_var′)))):
nth(cons(cons1_var′, cons2_var′), i) = x'

{1} x' = cons1_var′
{2} member(x', cons2_var′)

Repeatedly Skolemizing and flattening,

Case splitting on i!1 = 0,

we get 2 subgoals:

`nth_member.2.3.1:`

{-1} i' = 0
{-2} $i' < \text{length}(\text{cons}(\text{cons1\_var}', \text{cons2\_var}'))$
{-3} ∀ (x: T):
member(x, cons2_var′) ≡
($\exists$ (i: below(length(cons2_var′))): nth(cons2_var′, i) = x)
{-4} nth(cons(cons1_var′, cons2_var′), i') = x'

{1} x' = cons1_var′
{2} member(x', cons2_var′)

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `nth_member.2.3.1`.

`nth_member.2.3.2`:

{-1} $i' < \text{length}(\text{cons}(\text{cons1\_var}', \text{cons2\_var}'))$
{-2} $\forall\ (x\colon T)\colon$
$\quad \text{member}(x, \text{cons2\_var}') \equiv$
$\quad (\exists\ (i\colon \text{below}(\text{length}(\text{cons2\_var}')))\colon \text{nth}(\text{cons2\_var}', i) = x)$
{-3} $\text{nth}(\text{cons}(\text{cons1\_var}', \text{cons2\_var}'), i') = x'$

{1} $i' = 0$
{2} $x' = \text{cons1\_var}'$
{3} $\text{member}(x', \text{cons2\_var}')$

Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Instantiating the top quantifier in 2 with the terms: $i' - 1$,
This completes the proof of `nth_member.2.3.2`.
Q.E.D.

C.138.37 More_List_Props.flatten_TCC1

Terse proof for `flatten_TCC1`.
`flatten_TCC1`:

{1} $\forall\ (\text{ll}\colon \text{list}[\text{list}[T]], \text{car}\colon \text{list}[T], \text{cdr}\colon \text{list}[\text{list}[T]])\colon$
$\quad \text{ll} = \text{cons}(\text{car}, \text{cdr}) \supset \text{length}[\text{list}[T]](\text{cdr}) < \text{length}[\text{list}[T]](\text{ll})$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `flatten_TCC1`.
Q.E.D.

C.138.38 More_List_Props.member_append

Terse proof for `member_append`.
`member_append`:

{1} $\forall\ (x\colon T, l_1, l_2\colon \text{list}[T])\colon$
$\quad \text{member}(x, \text{append}(l_1, l_2)) \equiv \text{member}(x, l_1) \vee \text{member}(x, l_2)$

By induction on l_1, and by repeatedly rewriting and simplifying,
This completes the proof of `member_append`.
Q.E.D.

C.138.39 More_List_Props.member_flatten

Terse proof for `member_flatten`.
`member_flatten`:

{1} $\forall\ (x\colon T, \text{ll}\colon \text{list}[\text{list}[T]])\colon$
$\quad \text{member}(x, \text{flatten}(\text{ll})) \equiv (\exists\ (l\colon \text{list}[T])\colon \text{member}(l, \text{ll}) \wedge \text{member}(x, l))$

Inducting on ll on formula 1,
we get 2 subgoals:

`member_flatten.1:`

{1} ∀ (x: T):
member(x, flatten(null)) ≡ (∃ (l: list$[T]$): member(l, null) ∧ member(x, l))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `member_flatten.1`.
`member_flatten.2:`

{1} ∀ (cons1_var: list$[T]$, cons2_var: list$[$list$[T]]$):
(∀ (x: T):
member(x, flatten(cons2_var)) ≡
(∃ (l: list$[T]$): member(l, cons2_var) ∧ member(x, l)))
⊃
(∀ (x: T):
member(x, flatten(cons(cons1_var, cons2_var))) ≡
(∃ (l: list$[T]$): member(l, cons(cons1_var, cons2_var)) ∧ member(x, l)))

Repeatedly Skolemizing and flattening,
Expanding the definition of flatten,
Rewriting using member_append, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
`member_flatten.2.1:`

{-1} ∀ (x: T):
member(x, flatten(cons2_var′)) ≡
(∃ (l: list$[T]$): member(l, cons2_var′) ∧ member(x, l))
{-2} member(x', cons1_var′)
{1} ∃ (l: list$[T]$): member(l, cons(cons1_var′, cons2_var′)) ∧ member(x', l)

Instantiating the top quantifier in 1 with the terms: cons1_var′,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of member,
which is trivially true.
This completes the proof of `member_flatten.2.1`.
`member_flatten.2.2:`

{-1} ∀ (x: T):
member(x, flatten(cons2_var′)) ≡
(∃ (l: list$[T]$): member(l, cons2_var′) ∧ member(x, l))
{-2} member(x', flatten(cons2_var′))
{1} ∃ (l: list$[T]$): member(l, cons(cons1_var′, cons2_var′)) ∧ member(x', l)

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of member,
which is trivially true.
This completes the proof of `member_flatten.2.2`.

`member_flatten.2.3`:

$$
\begin{array}{ll}
\{\text{-1}\} & \forall\ (x\colon T)\colon \\
 & \quad \mathrm{member}(x,\ \mathrm{flatten}(\mathrm{cons2\_var}')) \equiv \\
 & \quad\ (\exists\ (l\colon \mathrm{list}[T])\colon \mathrm{member}(l,\ \mathrm{cons2\_var}') \wedge \mathrm{member}(x,\ l)) \\
\{\text{-2}\} & \exists\ (l\colon \mathrm{list}[T])\colon \mathrm{member}(l,\ \mathrm{cons}(\mathrm{cons1\_var}',\ \mathrm{cons2\_var}')) \wedge \mathrm{member}(x',\ l) \\
\hline
\{1\} & \mathrm{member}(x',\ \mathrm{cons1\_var}') \\
\{2\} & \mathrm{member}(x',\ \mathrm{flatten}(\mathrm{cons2\_var}'))
\end{array}
$$

Repeatedly Skolemizing and flattening,
Expanding the definition of member,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `member_flatten.2.3`.
Q.E.D.

C.138.40 More_List_Props.map2_TCC1

Terse proof for `map2_TCC1`.
`map2_TCC1`:

$$
\begin{array}{ll}
\{1\} & \forall\ (f\colon [T,\ T \to T],\ l_1,\ l_2\colon \mathrm{list}[T],\ h_1\colon T,\ t_1\colon \mathrm{list}[T])\colon \\
 & \quad l_1 = \mathrm{cons}(h_1,\ t_1) \supset \\
 & \quad\ (\forall\ (h_2\colon T,\ t_2\colon \mathrm{list}[T])\colon \\
 & \quad\quad l_2 = \mathrm{cons}(h_2,\ t_2) \supset \mathrm{length}[T](t_1) < \mathrm{length}[T](l_1))
\end{array}
$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `map2_TCC1`.
Q.E.D.

C.138.41 More_List_Props.map2_property

Terse proof for `map2_property`.
`map2_property`:

$$
\begin{array}{ll}
\{1\} & \forall\ (P\colon \mathrm{PRED}[T],\ f\colon [T,\ T \to T],\ l_1,\ l_2\colon \mathrm{list}[T])\colon \\
 & \quad (\forall\ (t_1,\ t_2\colon T)\colon P(f(t_1,\ t_2))) \supset \mathrm{every}(P)(\mathrm{map2}(f)(l_1,\ l_2))
\end{array}
$$

Inducting on l_1 on formula 1,
we get 2 subgoals:
`map2_property.1`:

$$
\begin{array}{ll}
\{1\} & \forall\ (P\colon \mathrm{PRED}[T],\ f\colon [T,\ T \to T],\ l_2\colon \mathrm{list}[T])\colon \\
 & \quad (\forall\ (t_1,\ t_2\colon T)\colon P(f(t_1,\ t_2))) \supset \mathrm{every}(P)(\mathrm{map2}(f)(\mathrm{null},\ l_2))
\end{array}
$$

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of map2,
which is trivially true.
This completes the proof of `map2_property.1`.

`map2_property.2:`

{1} $\forall$ (cons1_var: T, cons2_var: list$[T]$):
(∀ (P: PRED$[T]$, f: $[T, T \to T]$, l_2: list$[T]$):
(∀ (t_1, t_2: T): $P(f(t_1, t_2))$) ⊃ every(P)(map2(f)(cons2_var, l_2)))
⊃
(∀ (P: PRED$[T]$, f: $[T, T \to T]$, l_2: list$[T]$):
(∀ (t_1, t_2: T): $P(f(t_1, t_2))$) ⊃
every(P)(map2(f)(cons(cons1_var, cons2_var), l_2)))

Repeatedly Skolemizing and flattening,
Expanding the definition of map2,
Expanding the definition of every,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`map2_property.2.1:`

{-1} $\forall$ (P: PRED$[T]$, f: $[T, T \to T]$, l_2: list$[T]$):
(∀ (t_1, t_2: T): $P(f(t_1, t_2))$) ⊃ every(P)(map2(f)(cons2_var′, l_2))
{-2} $\forall$ (t_1, t_2: T): $P'(f'(t_1, t_2))$

{1} null?(l_2')
{2} every(P')(map2(f')(cons2_var′, cdr(l_2')))

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `map2_property.2.1`.
`map2_property.2.2:`

{-1} $\forall$ (P: PRED$[T]$, f: $[T, T \to T]$, l_2: list$[T]$):
(∀ (t_1, t_2: T): $P(f(t_1, t_2))$) ⊃ every(P)(map2(f)(cons2_var′, l_2))
{-2} $\forall$ (t_1, t_2: T): $P'(f'(t_1, t_2))$

{1} null?(l_2')
{2} $P'(f'$(cons1_var′, car(l_2')))

Instantiating quantified variables,
This completes the proof of `map2_property.2.2`.
Q.E.D.

C.139 Proofs for More_List_Props_2 (vfiasco-prelude.pvs)

C.139.1 More_List_Props_2.member_map

Terse proof for `member_map`.
`member_map:`

{1} $\forall$ (b: B, f: $[A \to B]$, l: list$[A]$):
member(b, map(f)(l)) $\equiv$ ($\exists$ (a: A): member(a, l) $\wedge$ $f(a) = b$)

By induction on l, and by repeatedly rewriting and simplifying,
This completes the proof of `member_map`.
Q.E.D.

C.140 Proofs for More_More_List_Props (vfiasco-prelude.pvs)

C.140.1 More_More_List_Props.list_all_length

Terse proof for `list_all_length`.

`list_all_length:`

{1} $\forall\ (n\colon\ \text{nat})\colon\ \exists\ (l\colon\ \text{list}[T])\colon\ \text{length}(l)\ =\ n$

Inducting on n on formula 1,
we get 2 subgoals:
`list_all_length.1:`

{1} $\exists\ (l\colon\ \text{list}[T])\colon\ \text{length}(l)\ =\ 0$

Instantiating the top quantifier in 1 with the terms: null,
Expanding the definition of length,
which is trivially true.
This completes the proof of `list_all_length.1`.
`list_all_length.2:`

{1} $\forall\ j\colon$
$(\exists\ (l\colon\ \text{list}[T])\colon\ \text{length}(l)\ =\ j)\ \supset$
$(\exists\ (l\colon\ \text{list}[T])\colon\ \text{length}(l)\ =\ j+1)$

Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: (cons(choose(LAMBDA (t: T): TRUE), l!1)),
we get 2 subgoals:
`list_all_length.2.1:`

{-1} $j'\ \geq\ 0$
{-2} $\text{length}(l')\ =\ j'$

{1} $\text{length}(\text{cons}(\text{choose}(\lambda\ (t\colon\ T)\colon\ \text{TRUE}),\ l'))\ =\ j'+1$

Expanding the definition of length,
which is trivially true.
This completes the proof of `list_all_length.2.1`.
`list_all_length.2.2:`

{-1} $j'\ \geq\ 0$
{-2} $\text{length}(l')\ =\ j'$

{1} $\text{nonempty?}[T](\lambda\ (t\colon\ T)\colon\ \text{TRUE})$

Expanding the definition of nonempty?,
Expanding the definition of empty?,
Expanding the definition of member,
which is trivially true.
This completes the proof of `list_all_length.2.2`.
Q.E.D.

C.141 Proofs for More_Relations (vfiasco-prelude.pvs)

C.141.1 More_Relations.zorn_lr

Terse proof for `zorn_lr`.

`zorn_lr`:

{1} $\forall\ (R\colon\ \mathrm{PRED}[[T,\ T]])\colon$
well_founded?$(R)\ \supset$
$\neg\ (\exists\ (f\colon\ [\mathrm{nat}\ \to\ T])\colon\ \forall\ (n\colon\ \mathrm{nat})\colon\ R(f(n+1),\ f(n)))$

Expanding the definition of well_founded?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: (LAMBDA (t: T): EXISTS (n: nat): f!1(n) = t),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`zorn_lr.1`:

{-1} $\exists\ (y\colon\ (\lambda\ (t\colon\ T)\colon\ \exists\ (n\colon\ \mathrm{nat})\colon\ f'(n)\ =\ t))\colon$
$\forall\ (x\colon\ (\lambda\ (t\colon\ T)\colon\ \exists\ (n\colon\ \mathrm{nat})\colon\ f'(n)\ =\ t))\colon\ (\neg\ R'(x,\ y))$
{-2} $\forall\ (n\colon\ \mathrm{nat})\colon\ R'(f'(1+n),\ f'(n))$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating quantified variables,
we get 3 subgoals:

`zorn_lr.1.1`:

{-1} $n'\ \geq\ 0$
{-2} $f'(n')\ =\ y'$
{-3} $R'(f'(1+n'),\ f'(n'))$
{1} $R'(f'(1+n'),\ y')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `zorn_lr.1.1`.

`zorn_lr.1.2`:

{-1} $n'\ \geq\ 0$
{-2} $f'(n')\ =\ y'$
{-3} $R'(f'(1+n'),\ f'(n'))$
{1} $\exists\ (n\colon\ \mathrm{nat})\colon\ f'(n)\ =\ f'(1+n')$

Instantiating quantified variables,
This completes the proof of `zorn_lr.1.2`.

`zorn_lr.1.3`:

{-1} $n'\ \geq\ 0$
{-2} $f'(n')\ =\ y'$
{-3} $\forall\ (x\colon\ (\lambda\ (t\colon\ T)\colon\ \exists\ (n\colon\ \mathrm{nat})\colon\ f'(n)\ =\ t))\colon\ (\neg\ R'(x,\ y'))$
{-4} $R'(f'(1+n'),\ f'(n'))$
{1} $\exists\ (n\colon\ \mathrm{nat})\colon\ f'(n)\ =\ f'(1+n')$

Instantiating quantified variables,
This completes the proof of `zorn_lr.1.3`.

`zorn_lr.2`:

$$\begin{array}{ll} \{\text{-1}\} & \forall\ (n\colon\ \text{nat})\colon\ R'(f'(1+n),\ f'(n)) \\ \hline \{1\} & \exists\ (y\colon\ T)\colon\ \exists\ (n\colon\ \text{nat})\colon\ f'(n) = y \end{array}$$

Instantiating the top quantifier in 1 with the terms: (f!1(0)),
Instantiating quantified variables,
This completes the proof of `zorn_lr.2`.
Q.E.D.

C.142 Proofs for More_Sets_Lemmas (vfiasco-prelude.pvs)

C.142.1 More_Sets_Lemmas.subset_equal

Terse proof for `subset_equal`.
`subset_equal`:

$$\begin{array}{ll} & \\ \hline \{1\} & \forall\ (a,\ b\colon\ \text{set}\lfloor T\rfloor)\colon\ a = b \supset (a \subseteq b) \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_equal`.
Q.E.D.

C.142.2 More_Sets_Lemmas.union_subset3

Terse proof for `union_subset3`.
`union_subset3`:

$$\begin{array}{ll} & \\ \hline \{1\} & \forall\ (a,\ b\colon\ \text{set}\lfloor T\rfloor)\colon\ (a \subseteq (b \cup a)) \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `union_subset3`.
Q.E.D.

C.142.3 More_Sets_Lemmas.subset_bigger_union_left

Terse proof for `subset_bigger_union_left`.
`subset_bigger_union_left`:

$$\begin{array}{ll} & \\ \hline \{1\} & \forall\ (a,\ b,\ c\colon\ \text{set}\lfloor T\rfloor)\colon\ (a \subseteq b) \supset (a \subseteq (b \cup c)) \end{array}$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_bigger_union_left`.
Q.E.D.

C.142.4 More_Sets_Lemmas.subset_bigger_union_right

Terse proof for `subset_bigger_union_right`.

subset_bigger_union_right:

$$\{1\} \quad \forall\ (a,\ b,\ c\colon \mathrm{set}[T])\colon\ (a \subseteq c)\ \supset\ (a \subseteq (b \cup c))$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of subset_bigger_union_right.
Q.E.D.

C.142.5 More_Sets_Lemmas.union_empty_left

Terse proof for union_empty_left.
union_empty_left:

$$\{1\} \quad \forall\ (a\colon \mathrm{set}[T])\colon\ (\mathrm{emptyset} \cup a)\ =\ a$$

Repeatedly Skolemizing and flattening,
Rewriting using union_commutative, matching in *,
Rewriting using union_empty, matching in *,
This completes the proof of union_empty_left.
Q.E.D.

C.142.6 More_Sets_Lemmas.union_left

Terse proof for union_left.
union_left:

$$\{1\} \quad \forall\ (a,\ b\colon \mathrm{set}[T],\ e\colon T)\colon\ a(e)\ \supset\ \mathrm{union}(a,\ b)(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of union_left.
Q.E.D.

C.142.7 More_Sets_Lemmas.union_right

Terse proof for union_right.
union_right:

$$\{1\} \quad \forall\ (a,\ b\colon \mathrm{set}[T],\ e\colon T)\colon\ b(e)\ \supset\ \mathrm{union}(a,\ b)(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of union_right.
Q.E.D.

C.142.8 More_Sets_Lemmas.subset_of_differences

Terse proof for subset_of_differences.
subset_of_differences:

$$\{1\} \quad \forall\ (a,\ b,\ c\colon \mathrm{set}[T])\colon\ (a \subseteq b)\ \supset\ ((c \setminus b) \subseteq (c \setminus a))$$

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `subset_of_differences`.
Q.E.D.

C.142.9 More_Sets_Lemmas.difference_disjoint_3

Terse proof for `difference_disjoint_3`.
`difference_disjoint_3:`

$$\{1\} \quad \forall\ (a,\ b\colon\ \mathrm{set}[T])\colon\ \mathrm{disjoint?}((b \setminus a),\ a)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `difference_disjoint_3`.
Q.E.D.

C.142.10 More_Sets_Lemmas.subset_singleton

Terse proof for `subset_singleton`.
`subset_singleton:`

$$\{1\} \quad \forall\ (a\colon\ \mathrm{set}[T],\ e\colon\ T)\colon\ (\mathrm{singleton}(e) \subseteq a)\ =\ a(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_singleton`.
Q.E.D.

C.142.11 More_Sets_Lemmas.subset_member

Terse proof for `subset_member`.
`subset_member:`

$$\{1\} \quad \forall\ (a,\ b\colon\ \mathrm{set}[T],\ e\colon\ T)\colon\ a(e)\ \wedge\ (a \subseteq b)\ \supset\ b(e)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `subset_member`.
Q.E.D.

C.142.12 More_Sets_Lemmas.disjoint_symmetric

Terse proof for `disjoint_symmetric`.
`disjoint_symmetric:`

$$\{1\} \quad \forall\ (a,\ b\colon\ \mathrm{set}[T])\colon\ \mathrm{disjoint?}(a,\ b)\ =\ \mathrm{disjoint?}(b,\ a)$$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `disjoint_symmetric`.
Q.E.D.

C.142.13 More_Sets_Lemmas.disjoint_mono

Terse proof for `disjoint_mono`.

`disjoint_mono:`

{1} $\forall\ (a_1,\ a_2,\ b_1,\ b_2\colon \text{set}[T])\colon$
$(a_1 \subseteq a_2)\ \wedge\ (b_1 \subseteq b_2)\ \wedge\ \text{disjoint?}(a_2,\ b_2)\ \supset\ \text{disjoint?}(a_1,\ b_1)$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `disjoint_mono`.
Q.E.D.

C.142.14 More_Sets_Lemmas.disjoint_subset_difference

Terse proof for `disjoint_subset_difference`.

`disjoint_subset_difference:`

{1} $\forall\ (a,\ b,\ c\colon \text{set}[T])\colon\ (a \subseteq b)\ \wedge\ \text{disjoint?}(a,\ c)\ \supset\ (a \subseteq (b \setminus c))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `disjoint_subset_difference`.
Q.E.D.

C.143 Proofs for Number_Props (vfiasco-prelude.pvs)

C.143.1 Number_Props.number_split

Terse proof for `number_split`.

`number_split:`

{1} $\forall\ (x\colon \text{int},\ a\colon \text{posnat})\colon\ x = a \times \text{ndiv}(x,\ a) + \text{rem}(a)(x)$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(x!1, a!1),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `number_split`.
Q.E.D.

C.143.2 Number_Props.number_split_less

Terse proof for `number_split_less`.

`number_split_less:`

{1} $\forall\ (a,\ b,\ p,\ q\colon \text{nat})\colon\ a < p\ \supset\ (a + b \times p < p \times q \equiv b < q)$

Repeatedly Skolemizing and flattening,
Applying both_sides_times_pos_le1
Instantiating the top quantifier in -1 with the terms: p', $b' + 1$, q',
we get 2 subgoals:

`number_split_less.1`:

$$\begin{array}{ll}\{\text{-1}\} & b' + 1 \times p' \leq q' \times p' \equiv b' + 1 \leq q' \\ \{\text{-2}\} & a' \geq 0 \\ \{\text{-3}\} & b' \geq 0 \\ \{\text{-4}\} & p' \geq 0 \\ \{\text{-5}\} & q' \geq 0 \\ \{\text{-6}\} & a' < p' \\ \hline \{1\} & (a' + b' \times p' < p' \times q' \equiv b' < q') \end{array}$$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Applying both_sides_times_pos_lt1
Instantiating the top quantifier in -1 with the terms: p', b', q',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `number_split_less.1`.
`number_split_less.2`:

$$\begin{array}{ll}\{\text{-1}\} & a' \geq 0 \\ \{\text{-2}\} & b' \geq 0 \\ \{\text{-3}\} & p' \geq 0 \\ \{\text{-4}\} & q' \geq 0 \\ \{\text{-5}\} & a' < p' \\ \hline \{1\} & p' > 0 \\ \{2\} & (a' + b' \times p' < p' \times q' \equiv b' < q') \end{array}$$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `number_split_less.2`.
Q.E.D.

C.143.3 Number_Props.bit_split_less

Terse proof for `bit_split_less`.
`bit_split_less`:

$$\begin{array}{ll}\{1\} & \forall\ (q,\ r,\ j,\ k\colon \text{nat}): \\ & \quad r < \text{expt}(2,\ k) \supset \\ & \quad\ (q \times \text{expt}(2,\ k) + r < \text{expt}(2,\ k + j) \equiv q < \text{expt}(2,\ j)) \end{array}$$

Repeatedly Skolemizing and flattening,
Applying number_split_less
Instantiating the top quantifier in -1 with the terms: r', q', expt(2, k'), expt(2, j'),
Rewriting using expt_plus_aux, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `bit_split_less`.
Q.E.D.

C.143.4 Number_Props.rem_def_pos

Terse proof for `rem_def_pos`.
`rem_def_pos`:

$$\begin{array}{ll}\{1\} & \forall\ (b\colon \text{posnat},\ x\colon \text{nat}): \\ & \quad \forall\ (r\colon \text{mod}(b))\colon \text{rem}(b)(x) = r \equiv (\exists\ (q\colon \text{nat})\colon x = b \times q + r) \end{array}$$

Repeatedly Skolemizing and flattening,
Using lemma rem_def,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`rem_def_pos.1:`

{-1}	$\mathrm{rem}(b')(x') = r'$
{-2}	$\exists\ q\colon\ x' = b' \times q + r'$
{-3}	$r' < b'$
{-4}	$b' > 0$
{-5}	$x' \geq 0$
{1}	$\exists\ (q\colon\ \mathrm{nat})\colon\ x' = b' \times q + r'$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Using lemma both_sides_times_pos_le2,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_def_pos.1`.
`rem_def_pos.2:`

{-1}	$r' < b'$
{-2}	$b' > 0$
{-3}	$x' \geq 0$
{-4}	$\exists\ (q\colon\ \mathrm{nat})\colon\ x' = b' \times q + r'$
{1}	$\mathrm{rem}(b')(x') = r'$
{2}	$\exists\ q\colon\ x' = b' \times q + r'$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of `rem_def_pos.2`.
Q.E.D.

C.143.5 Number_Props.rem_rem

Terse proof for `rem_rem`.
`rem_rem:`

{1}	$\forall\ (a,\ b\colon\ \mathrm{posnat},\ x\colon\ \mathrm{int})\colon\ \mathrm{divides}(a,\ b) \supset \mathrm{rem}(a)(\mathrm{rem}(b)(x)) = \mathrm{rem}(a)(x)$

Repeatedly Skolemizing and flattening,
Applying rem_def
Instantiating the top quantifier in -1 with the terms: a', x', $\mathrm{rem}(a')(\mathrm{rem}(b')(x'))$,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: 2,
Adding type constraints for rem(a!1)(rem(b!1)(x!1)),
Adding type constraints for rem(b!1)(x!1),
Expanding the definition of divides,
Repeatedly Skolemizing and flattening,
Replacing using formula -7,
Instantiating the top quantifier in 1 with the terms: $q'' + x'' \times q'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_rem`.
Q.E.D.

C.143.6 Number_Props.divides_times

Terse proof for `divides_times`.

`divides_times:`

{1} $\forall$ (i: posnat, j, k: int):
divides($i \times j$, k) $\equiv$ divides(i, k) $\wedge$ divides(j, ndiv(k, i))

Repeatedly Skolemizing and flattening,
Splitting conjunctions,
we get 2 subgoals:

`divides_times.1:`

{-1} $i' > 0$
{-2} integer_pred(j')
{-3} integer_pred(k')
{1} divides($i' \times j'$, k') $\supset$ divides(i', k') $\wedge$ divides(j', ndiv(k', i'))

Case splitting on divides(i!1, k!1),
we get 2 subgoals:

`divides_times.1.1:`

{-1} divides(i', k')
{-2} $i' > 0$
{-3} integer_pred(j')
{-4} integer_pred(k')
{1} divides($i' \times j'$, k') $\supset$ divides(i', k') $\wedge$ divides(j', ndiv(k', i'))

Applying propositional simplification,
Adding type constraints for ndiv(k!1, i!1),
Case splitting on rem(i!1)(k!1) = 0,
we get 2 subgoals:

`divides_times.1.1.1:`

{-1} rem(i')(k') = 0
{-2} $k' =$ rem(i')(k') + $i' \times$ ndiv(k', i')
{-3} divides(i', k')
{-4} divides($i' \times j'$, k')
{-5} $i' > 0$
{-6} integer_pred(j')
{-7} integer_pred(k')
{1} divides(j', ndiv(k', i'))

Replacing using formula -1,
Hiding formulas: -1, -3,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of divides,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: x',
Applying both_sides_times1
Instantiating the top quantifier in -1 with the terms: i', ndiv(k', i'), $j' \times x'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `divides_times.1.1.1`.

`divides_times.1.1.2:`

{-1}	$k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k',\ i')$
{-2}	$\text{divides}(i',\ k')$
{-3}	$\text{divides}(i' \times j',\ k')$
{-4}	$i' > 0$
{-5}	$\text{integer\_pred}(j')$
{-6}	$\text{integer\_pred}(k')$
{1}	$\text{rem}(i')(k') = 0$
{2}	$\text{divides}(j',\ \text{ndiv}(k',\ i'))$

Rewriting using rem_def2, matching in *,
This completes the proof of `divides_times.1.1.2`.
`divides_times.1.2:`

{-1}	$i' > 0$
{-2}	$\text{integer\_pred}(j')$
{-3}	$\text{integer\_pred}(k')$
{1}	$\text{divides}(i',\ k')$
{2}	$\text{divides}(i' \times j',\ k') \supset \text{divides}(i',\ k') \wedge \text{divides}(j',\ \text{ndiv}(k',\ i'))$

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of divides,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: $x' \times j'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `divides_times.1.2`.
`divides_times.2:`

{-1}	$i' > 0$
{-2}	$\text{integer\_pred}(j')$
{-3}	$\text{integer\_pred}(k')$
{1}	$\text{divides}(i',\ k') \wedge \text{divides}(j',\ \text{ndiv}(k',\ i')) \supset \text{divides}(i' \times j',\ k')$

Applying disjunctive simplification to flatten sequent,
Adding type constraints for ndiv(k!1, i!1),
Case splitting on rem(i!1)(k!1) = 0,
we get 2 subgoals:
`divides_times.2.1:`

{-1}	$\text{rem}(i')(k') = 0$
{-2}	$k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k',\ i')$
{-3}	$\text{divides}(i',\ k')$
{-4}	$\text{divides}(j',\ \text{ndiv}(k',\ i'))$
{-5}	$i' > 0$
{-6}	$\text{integer\_pred}(j')$
{-7}	$\text{integer\_pred}(k')$
{1}	$\text{divides}(i' \times j',\ k')$

Replacing using formula -1,
Hiding formulas: -1, -3,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of divides,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: x',
Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `divides_times.2.1`.
`divides_times.2.2`:

{-1}	$k' = \text{rem}(i')(k') + i' \times \text{ndiv}(k', i')$
{-2}	$\text{divides}(i', k')$
{-3}	$\text{divides}(j', \text{ndiv}(k', i'))$
{-4}	$i' > 0$
{-5}	$\text{integer\_pred}(j')$
{-6}	$\text{integer\_pred}(k')$
{1}	$\text{rem}(i')(k') = 0$
{2}	$\text{divides}(i' \times j', k')$

Rewriting using rem_def2, matching in *,
This completes the proof of `divides_times.2.2`.
Q.E.D.

C.143.7 Number_Props.ndiv_floor

Terse proof for `ndiv_floor`.
`ndiv_floor`:

{1}	$\forall$ (a: int, b: posnat): $\text{ndiv}(a, b) = \text{floor}(a/b)$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(a!1, b!1),
Using lemma rem_floor,
Applying unique_quotient
Instantiating the top quantifier in -1 with the terms: b', $\text{ndiv}(a', b')$, $\text{floor}(a'/b')$, $\text{rem}(b')(a')$, $\text{rem}(b')(a')$,
Simplifying, rewriting, and recording with decision procedures,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_floor`.
Q.E.D.

C.143.8 Number_Props.ndiv_self

Terse proof for `ndiv_self`.
`ndiv_self`:

{1}	$\forall$ (a: posnat): $\text{ndiv}(a, a) = 1$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(a!1, a!1),
Rewriting using rem_self, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Applying both_sides_times2
Instantiating the top quantifier in -1 with the terms: a', 1, $\text{ndiv}(a', a')$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_self`.
Q.E.D.

C.143.9 Number_Props.ndiv_plus_1

Terse proof for `ndiv_plus_1`.

`ndiv_plus_1:`

{1}	$\forall$ (a, b: int, c: posnat): divides(c, a) $\supset$ ndiv($a + b$, c) = ndiv(a, c) + ndiv(b, c)

Repeatedly Skolemizing and flattening,
Applying rem_def2
Instantiating the top quantifier in -1 with the terms: c', a', 0,
Simplifying, rewriting, and recording with decision procedures,
Adding type constraints for ndiv(a!1 + b!1, c!1),
Applying rem_sum1
Instantiating the top quantifier in -1 with the terms: c', a', b',
Replacing using formula -3,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Adding type constraints for ndiv(a!1, c!1),
Adding type constraints for ndiv(b!1, c!1),
Case splitting on c!1 * (ndiv(a!1, c!1) + ndiv(b!1, c!1)) = c!1 * ndiv(a!1 + b!1, c!1),
we get 2 subgoals:

`ndiv_plus_1.1:`

{-1}	$c' \times (\text{ndiv}(a', c') + \text{ndiv}(b', c')) = c' \times \text{ndiv}(a' + b', c')$
{-2}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-3}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-4}	$a' + b' = \text{rem}(c')(b') + c' \times \text{ndiv}(a' + b', c')$
{-5}	$\text{rem}(c')(a') = 0$
{-6}	integer_pred(a')
{-7}	integer_pred(b')
{-8}	$c' > 0$
{-9}	divides(c', a')
{1}	$\text{ndiv}(a' + b', c') = \text{ndiv}(a', c') + \text{ndiv}(b', c')$

Applying both_sides_times2
Instantiating the top quantifier in -1 with the terms: c', (ndiv(a', c') + ndiv(b', c')), ndiv($a' + b'$, c'),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_plus_1.1`.

`ndiv_plus_1.2:`

{-1}	$b' = \text{rem}(c')(b') + c' \times \text{ndiv}(b', c')$
{-2}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a', c')$
{-3}	$a' + b' = \text{rem}(c')(b') + c' \times \text{ndiv}(a' + b', c')$
{-4}	$\text{rem}(c')(a') = 0$
{-5}	integer_pred(a')
{-6}	integer_pred(b')
{-7}	$c' > 0$
{-8}	divides(c', a')
{1}	$c' \times (\text{ndiv}(a', c') + \text{ndiv}(b', c')) = c' \times \text{ndiv}(a' + b', c')$
{2}	$\text{ndiv}(a' + b', c') = \text{ndiv}(a', c') + \text{ndiv}(b', c')$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_plus_1.2`.

Q.E.D.

C.143.10 Number_Props.ndiv_plus_2

Terse proof for `ndiv_plus_2`.

`ndiv_plus_2:`

$$\frac{}{\{1\}\quad \forall\ (a,\ b\colon \text{int},\ c\colon \text{posnat})\colon \text{divides}(c,\ b) \supset \text{ndiv}(a+b,\ c) = \text{ndiv}(a,\ c) + \text{ndiv}(b,\ c)}$$

Repeatedly Skolemizing and flattening,

Using lemma ndiv_plus_1,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_plus_2`.

Q.E.D.

C.143.11 Number_Props.ndiv_plus_mod

Terse proof for `ndiv_plus_mod`.

`ndiv_plus_mod:`

$$\frac{}{\{1\}\quad \forall\ (a\colon \text{int},\ c\colon \text{posnat},\ b\colon \text{mod}(c))\colon \text{divides}(c,\ a) \supset \text{ndiv}(a+b,\ c) = \text{ndiv}(a,\ c)}$$

Repeatedly Skolemizing and flattening,

Adding type constraints for ndiv(a!1, c!1),

Case splitting on rem(c!1)(a!1) = 0,

we get 2 subgoals:

`ndiv_plus_mod.1:`

$$\frac{\begin{array}{ll}\{\text{-}1\} & \text{rem}(c')(a') = 0\\ \{\text{-}2\} & a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a',\ c')\\ \{\text{-}3\} & \text{integer\_pred}(a')\\ \{\text{-}4\} & c' > 0\\ \{\text{-}5\} & b' < c'\\ \{\text{-}6\} & \text{divides}(c',\ a')\end{array}}{\{1\}\quad \text{ndiv}(a'+b',\ c') = \text{ndiv}(a',\ c')}$$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

Adding type constraints for ndiv(a!1 + b!1, c!1),

Case splitting on rem(c!1)(a!1 + b!1) = b!1,

we get 2 subgoals:

`ndiv_plus_mod.1.1:`

{-1}	$\text{rem}(c')(a' + b') = b'$
{-2}	$a' + b' = \text{rem}(c')(a' + b') + c' \times \text{ndiv}(a' + b',\ c')$
{-3}	$a' = c' \times \text{ndiv}(a',\ c')$
{-4}	$\text{integer\_pred}(a')$
{-5}	$c' > 0$
{-6}	$b' < c'$
{-7}	$\text{divides}(c',\ a')$
{1}	$\text{ndiv}(a' + b',\ c') = \text{ndiv}(a',\ c')$

Replacing using formula -1,
Hiding formulas: -1,
Simplifying, rewriting, and recording with decision procedures,
Applying both_sides_times2
Instantiating the top quantifier in -1 with the terms: c', $\text{ndiv}(a' + b', c')$, $\text{ndiv}(a', c')$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_plus_mod.1.1`.
`ndiv_plus_mod.1.2:`

{-1}	$a' + b' = \text{rem}(c')(a' + b') + c' \times \text{ndiv}(a' + b',\ c')$
{-2}	$a' = c' \times \text{ndiv}(a',\ c')$
{-3}	$\text{integer\_pred}(a')$
{-4}	$c' > 0$
{-5}	$b' < c'$
{-6}	$\text{divides}(c',\ a')$
{1}	$\text{rem}(c')(a' + b') = b'$
{2}	$\text{ndiv}(a' + b',\ c') = \text{ndiv}(a',\ c')$

Hiding formulas: -1, -2, 2,
Rewriting using rem_def, matching in *,
Expanding the definition of divides,
which is trivially true.
This completes the proof of `ndiv_plus_mod.1.2`.
`ndiv_plus_mod.2:`

{-1}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a',\ c')$
{-2}	$\text{integer\_pred}(a')$
{-3}	$c' > 0$
{-4}	$b' < c'$
{-5}	$\text{divides}(c',\ a')$
{1}	$\text{rem}(c')(a') = 0$
{2}	$\text{ndiv}(a' + b',\ c') = \text{ndiv}(a',\ c')$

Rewriting using rem_def2, matching in *,
This completes the proof of `ndiv_plus_mod.2`.
Q.E.D.

C.143.12 Number_Props.ndiv_plus_mod_2

Terse proof for `ndiv_plus_mod_2`.

ndiv_plus_mod_2:

{1} $\forall$ (a: int, c: posnat, b_1, b_2: mod(c)):
divides(c, a) $\supset$ ndiv($a + b_1$, c) = ndiv($a + b_2$, c)

Repeatedly Skolemizing and flattening,
Applying ndiv_plus_mod
Instantiating (with copying) the top quantifier in -1 with the terms: a', c', b'_1,
Instantiating the top quantifier in -1 with the terms: a', c', b'_2,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of ndiv_plus_mod_2.
Q.E.D.

C.143.13 Number_Props.ndiv_minus_2

Terse proof for ndiv_minus_2.
ndiv_minus_2:

{1} $\forall$ (a, b: int, c: posnat):
divides(c, b) $\supset$ ndiv($a - b$, c) = ndiv(a, c) $-$ ndiv(b, c)

Repeatedly Skolemizing and flattening,
Applying ndiv_plus_2
Instantiating the top quantifier in -1 with the terms: a', $-b'$, c',
Case splitting on divides(c!1, -b!1),
we get 2 subgoals:
ndiv_minus_2.1:

{-1} divides(c', $-b'$)
{-2} divides(c', $-b'$) $\supset$
ndiv($a' + -b'$, c') = ndiv(a', c') + ndiv($-b'$, c')
{-3} integer_pred(a')
{-4} integer_pred(b')
{-5} $c' > 0$
{-6} divides(c', b')
{1} ndiv($a' - b'$, c') = ndiv(a', c') $-$ ndiv(b', c')

Simplifying, rewriting, and recording with decision procedures,
Case splitting on -ndiv(b!1, c!1) = ndiv(-b!1, c!1),
we get 2 subgoals:
ndiv_minus_2.1.1:

{-1} $-$ndiv(b', c') = ndiv($-b'$, c')
{-2} divides(c', $-b'$)
{-3} ndiv($-b' + a'$, c') = ndiv($-b'$, c') + ndiv(a', c')
{-4} integer_pred(a')
{-5} integer_pred(b')
{-6} $c' > 0$
{-7} divides(c', b')
{1} ndiv($a' - b'$, c') = ndiv(a', c') $-$ ndiv(b', c')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of ndiv_minus_2.1.1.

`ndiv_minus_2.1.2:`

{-1}	divides(c', $-b'$)
{-2}	ndiv($-b' + a'$, c') = ndiv($-b'$, c') + ndiv(a', c')
{-3}	integer_pred(a')
{-4}	integer_pred(b')
{-5}	$c' > 0$
{-6}	divides(c', b')
{1}	$-$ndiv(b', c') = ndiv($-b'$, c')
{2}	ndiv($a' - b'$, c') = ndiv(a', c') $-$ ndiv(b', c')

Hiding formulas: -2, 2,

Adding type constraints for ndiv(b!1, c!1),

Adding type constraints for ndiv(-b!1, c!1),

Case splitting on rem(c!1)(-b!1) = 0,

we get 2 subgoals:

`ndiv_minus_2.1.2.1:`

{-1}	rem(c')($-b'$) = 0
{-2}	$-b'$ = rem(c')($-b'$) + $c' \times$ ndiv($-b'$, c')
{-3}	b' = rem(c')(b') + $c' \times$ ndiv(b', c')
{-4}	divides(c', $-b'$)
{-5}	integer_pred(a')
{-6}	integer_pred(b')
{-7}	$c' > 0$
{-8}	divides(c', b')
{1}	$-$ndiv(b', c') = ndiv($-b'$, c')

Case splitting on rem(c!1)(b!1) = 0,

we get 2 subgoals:

`ndiv_minus_2.1.2.1.1:`

{-1}	rem(c')(b') = 0
{-2}	rem(c')($-b'$) = 0
{-3}	$-b'$ = rem(c')($-b'$) + $c' \times$ ndiv($-b'$, c')
{-4}	b' = rem(c')(b') + $c' \times$ ndiv(b', c')
{-5}	divides(c', $-b'$)
{-6}	integer_pred(a')
{-7}	integer_pred(b')
{-8}	$c' > 0$
{-9}	divides(c', b')
{1}	$-$ndiv(b', c') = ndiv($-b'$, c')

Replacing using formula -1,

Replacing using formula -2,

Hiding formulas: -1, -2,

Simplifying, rewriting, and recording with decision procedures,

Applying both_sides_times2

Instantiating the top quantifier in -1 with the terms: c', $-$ndiv(b', c'), ndiv($-b'$, c'),

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_minus_2.1.2.1.1`.

`ndiv_minus_2.1.2.1.2:`

{-1}	$\mathrm{rem}(c')(-b') = 0$
{-2}	$-b' = \mathrm{rem}(c')(-b') + c' \times \mathrm{ndiv}(-b', c')$
{-3}	$b' = \mathrm{rem}(c')(b') + c' \times \mathrm{ndiv}(b', c')$
{-4}	$\mathrm{divides}(c', -b')$
{-5}	$\mathrm{integer\_pred}(a')$
{-6}	$\mathrm{integer\_pred}(b')$
{-7}	$c' > 0$
{-8}	$\mathrm{divides}(c', b')$
{1}	$\mathrm{rem}(c')(b') = 0$
{2}	$-\mathrm{ndiv}(b', c') = \mathrm{ndiv}(-b', c')$

Rewriting using rem_def2, matching in *,
This completes the proof of `ndiv_minus_2.1.2.1.2`.
`ndiv_minus_2.1.2.2:`

{-1}	$-b' = \mathrm{rem}(c')(-b') + c' \times \mathrm{ndiv}(-b', c')$
{-2}	$b' = \mathrm{rem}(c')(b') + c' \times \mathrm{ndiv}(b', c')$
{-3}	$\mathrm{divides}(c', -b')$
{-4}	$\mathrm{integer\_pred}(a')$
{-5}	$\mathrm{integer\_pred}(b')$
{-6}	$c' > 0$
{-7}	$\mathrm{divides}(c', b')$
{1}	$\mathrm{rem}(c')(-b') = 0$
{2}	$-\mathrm{ndiv}(b', c') = \mathrm{ndiv}(-b', c')$

Rewriting using rem_def2, matching in *,
This completes the proof of `ndiv_minus_2.1.2.2`.
`ndiv_minus_2.2:`

{-1}	$\mathrm{divides}(c', -b') \supset$ $\mathrm{ndiv}(a' + -b', c') = \mathrm{ndiv}(a', c') + \mathrm{ndiv}(-b', c')$
{-2}	$\mathrm{integer\_pred}(a')$
{-3}	$\mathrm{integer\_pred}(b')$
{-4}	$c' > 0$
{-5}	$\mathrm{divides}(c', b')$
{1}	$\mathrm{divides}(c', -b')$
{2}	$\mathrm{ndiv}(a' - b', c') = \mathrm{ndiv}(a', c') - \mathrm{ndiv}(b', c')$

Hiding formulas: -1, 2,
Expanding the definition of divides,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in 1 with the terms: $-x'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_minus_2.2`.
Q.E.D.

C.143.14 Number_Props.ndiv_times_divident_1

Terse proof for `ndiv_times_divident_1`.
`ndiv_times_divident_1:`

{1}	$\forall$ (a, b: int, c: posnat): $\mathrm{divides}(c, a) \supset \mathrm{ndiv}(a \times b, c) = \mathrm{ndiv}(a, c) \times b$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(a!1 * b!1, c!1),
Adding type constraints for ndiv(a!1, c!1),
Case splitting on rem(c!1)(a!1) = 0,
we get 2 subgoals:
`ndiv_times_divident_1.1:`

{-1}	$\mathrm{rem}(c')(a') = 0$
{-2}	$a' = \mathrm{rem}(c')(a') + c' \times \mathrm{ndiv}(a', c')$
{-3}	$a' \times b' = \mathrm{rem}(c')(a' \times b') + c' \times \mathrm{ndiv}(a' \times b', c')$
{-4}	$\mathrm{integer\_pred}(a')$
{-5}	$\mathrm{integer\_pred}(b')$
{-6}	$c' > 0$
{-7}	$\mathrm{divides}(c', a')$
{1}	$\mathrm{ndiv}(a' \times b', c') = \mathrm{ndiv}(a', c') \times b'$

Replacing using formula -1,
Hiding formulas: -1,
Case splitting on rem(c!1)(a!1 * b!1) = 0,
we get 2 subgoals:
`ndiv_times_divident_1.1.1:`

{-1}	$\mathrm{rem}(c')(a' \times b') = 0$
{-2}	$a' = 0 + c' \times \mathrm{ndiv}(a', c')$
{-3}	$a' \times b' = \mathrm{rem}(c')(a' \times b') + c' \times \mathrm{ndiv}(a' \times b', c')$
{-4}	$\mathrm{integer\_pred}(a')$
{-5}	$\mathrm{integer\_pred}(b')$
{-6}	$c' > 0$
{-7}	$\mathrm{divides}(c', a')$
{1}	$\mathrm{ndiv}(a' \times b', c') = \mathrm{ndiv}(a', c') \times b'$

Replacing using formula -1,
Hiding formulas: -1,
Simplifying, rewriting, and recording with decision procedures,
Applying both_sides_times2
Instantiating the top quantifier in -1 with the terms: c', $\mathrm{ndiv}(a', c') \times b'$, $\mathrm{ndiv}(a' \times b', c')$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_times_divident_1.1.1`.
`ndiv_times_divident_1.1.2:`

{-1}	$a' = 0 + c' \times \mathrm{ndiv}(a', c')$
{-2}	$a' \times b' = \mathrm{rem}(c')(a' \times b') + c' \times \mathrm{ndiv}(a' \times b', c')$
{-3}	$\mathrm{integer\_pred}(a')$
{-4}	$\mathrm{integer\_pred}(b')$
{-5}	$c' > 0$
{-6}	$\mathrm{divides}(c', a')$
{1}	$\mathrm{rem}(c')(a' \times b') = 0$
{2}	$\mathrm{ndiv}(a' \times b', c') = \mathrm{ndiv}(a', c') \times b'$

Rewriting using rem_def2, matching in *,
Rewriting using divides_prod1, matching in *,
This completes the proof of `ndiv_times_divident_1.1.2`.

`ndiv_times_divident_1.2:`

{-1} $a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a',\ c')$
{-2} $a' \times b' = \text{rem}(c')(a' \times b') + c' \times \text{ndiv}(a' \times b',\ c')$
{-3} integer_pred(a')
{-4} integer_pred(b')
{-5} $c' > 0$
{-6} divides(c', a')

{1} $\text{rem}(c')(a') = 0$
{2} $\text{ndiv}(a' \times b',\ c') = \text{ndiv}(a',\ c') \times b'$

Rewriting using rem_def2, matching in *,
This completes the proof of `ndiv_times_divident_1.2`.
Q.E.D.

C.143.15 Number_Props.ndiv_times_divident_2

Terse proof for `ndiv_times_divident_2`.
`ndiv_times_divident_2:`

{1} $\forall$ (a, b: int, c: posnat):
divides(c, b) $\supset$ $\text{ndiv}(a \times b,\ c) = a \times \text{ndiv}(b,\ c)$

Repeatedly Skolemizing and flattening,
Applying ndiv_times_divident_1
Instantiating the top quantifier in -1 with the terms: b', a', c',
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_times_divident_2`.
Q.E.D.

C.143.16 Number_Props.rem_prod_3

Terse proof for `rem_prod_3`.
`rem_prod_3:`

{1} $\forall$ (b_1, b_2: posnat, x: int):
$\text{rem}(b_1 \times b_2)(x) = \text{rem}(b_1)(x) + \text{rem}(b_2)(\text{ndiv}(x,\ b_1)) \times b_1$

Installing automatic rewrites from: ndiv_self divides_reflexive rem_def2
Repeatedly Skolemizing and flattening,
Rewriting using rem_def2, matching in *,
we get 2 subgoals:
`rem_prod_3.1:`

{-1} $b'_1 > 0$
{-2} $b'_2 > 0$
{-3} integer_pred(x')

{1} divides($b'_1 \times b'_2$,
$-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x',\ b'_1)) \times b'_1 + x'$)

Rewriting using divides_times, matching in *,
Applying propositional simplification,
we get 2 subgoals:

`rem_prod_3.1.1:`

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	integer_pred(x')
{1}	divides(b'_1, $-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x'$)

Applying divides_diff
Instantiating the top quantifier in -1 with the terms: $\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$, $x' - \text{rem}(b'_1)(x')$, b'_1,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: 2,
Applying propositional simplification,
we get 2 subgoals:
`rem_prod_3.1.1.1:`

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	integer_pred(x')
{1}	divides(b'_1, $x' - \text{rem}(b'_1)(x')$)

Using lemma rem_def2,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_prod_3.1.1.1`.
`rem_prod_3.1.1.2:`

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	integer_pred(x')
{1}	divides(b'_1, $\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$)

Using lemma divides_prod2,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_prod_3.1.1.2`.
`rem_prod_3.1.2:`

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	integer_pred(x')
{1}	divides(b'_2, $\text{ndiv}(-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x', b'_1)$)

Applying ndiv_minus_2
Instantiating the top quantifier in -1 with the terms: $x' - \text{rem}(b'_1)(x')$, $\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$, b'_1,
Splitting conjunctions,
we get 2 subgoals:
`rem_prod_3.1.2.1:`

{-1}	$\text{ndiv}(x' - \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1, b'_1) = \text{ndiv}(x' - \text{rem}(b'_1)(x'), b'_1) - \text{ndiv}(\text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1, b'_1)$
{-2}	$b'_1 > 0$
{-3}	$b'_2 > 0$
{-4}	integer_pred(x')
{1}	divides(b'_2, $\text{ndiv}(-1 \times \text{rem}(b'_1)(x') - \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 + x', b'_1)$)

Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Applying ndiv_times_divident_2
Instantiating the top quantifier in -1 with the terms: $\mathrm{rem}(b_2')(\mathrm{ndiv}(x', b_1'))$, b_1', b_1',
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Hiding formulas: -1,
Applying ndiv_plus_mod
Instantiating the top quantifier in -1 with the terms: $x' - \mathrm{rem}(b_1')(x')$, b_1', $\mathrm{rem}(b_1')(x')$,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`rem_prod_3.1.2.1.1:`

{-1}	$\mathrm{ndiv}(x',\ b_1') = \mathrm{ndiv}(x' - \mathrm{rem}(b_1')(x'),\ b_1')$
{-2}	$b_1' > 0$
{-3}	$b_2' > 0$
{-4}	$\mathrm{integer\_pred}(x')$
{1}	$\mathrm{divides}(b_2',\ \mathrm{ndiv}(x' - \mathrm{rem}(b_1')(x'),\ b_1') - \mathrm{rem}(b_2')(\mathrm{ndiv}(x',\ b_1')))$

Replacing using formula -1,
Hiding formulas: -1,
Adding type constraints for rem(b2!1)(ndiv(x!1, b1!1)),
Repeatedly Skolemizing and flattening,
Expanding the definition of divides,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_prod_3.1.2.1.1`.
`rem_prod_3.1.2.1.2:`

{-1}	$b_1' > 0$
{-2}	$b_2' > 0$
{-3}	$\mathrm{integer\_pred}(x')$
{1}	$\mathrm{divides}(b_1',\ x' - \mathrm{rem}(b_1')(x'))$
{2}	$\mathrm{divides}(b_2',\ \mathrm{ndiv}(x' - \mathrm{rem}(b_1')(x'),\ b_1') - \mathrm{rem}(b_2')(\mathrm{ndiv}(x',\ b_1')))$

Hiding formulas: 2,
Adding type constraints for rem(b1!1)(x!1),
Repeatedly Skolemizing and flattening,
Expanding the definition of divides,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_prod_3.1.2.1.2`.
`rem_prod_3.1.2.2:`

{-1}	$b_1' > 0$
{-2}	$b_2' > 0$
{-3}	$\mathrm{integer\_pred}(x')$
{1}	$\mathrm{divides}(b_1',\ \mathrm{rem}(b_2')(\mathrm{ndiv}(x',\ b_1')) \times b_1')$
{2}	$\mathrm{divides}(b_2', \mathrm{ndiv}(-1 \times \mathrm{rem}(b_1')(x') - \mathrm{rem}(b_2')(\mathrm{ndiv}(x',\ b_1')) \times b_1' + x', b_1'))$

Hiding formulas: 2,
Using lemma divides_prod2,

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_prod_3.1.2.2`.
`rem_prod_3.2:`

{-1}	$b'_1 > 0$
{-2}	$b'_2 > 0$
{-3}	integer_pred(x')
{1}	$\text{rem}(b'_1)(x') + \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1 < b'_1 \times b'_2$
{2}	$\text{rem}(b'_1 \times b'_2)(x') =$ $\text{rem}(b'_1)(x') + \text{rem}(b'_2)(\text{ndiv}(x', b'_1)) \times b'_1$

Hiding formulas: 2,
Applying number_split_less
Instantiating the top quantifier in -1 with the terms: $\text{rem}(b'_1)(x')$, $\text{rem}(b'_2)(\text{ndiv}(x', b'_1))$, b'_1, b'_2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `rem_prod_3.2`.
Q.E.D.

C.143.17 Number_Props.ndiv_times_2

Terse proof for `ndiv_times_2`.
`ndiv_times_2:`

{1}	$\forall$ (a: int, b, c: posnat): $\text{ndiv}(a, b \times c) = \text{ndiv}(\text{ndiv}(a, b), c)$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(a!1, b!1),
Adding type constraints for ndiv(a!1, b!1 * c!1),
Adding type constraints for ndiv(ndiv(a!1, b!1), c!1),
Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using rem_prod_3, matching in *,
Hiding formulas: -1,
Applying both_sides_times1
Instantiating the top quantifier in -1 with the terms: $b' \times c'$, $\text{ndiv}(a', b' \times c')$, $\text{ndiv}(\text{ndiv}(a', b'), c')$,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ndiv_times_2`.
Q.E.D.

C.143.18 Number_Props.both_sides_ndiv_lt1

Terse proof for `both_sides_ndiv_lt1`.
`both_sides_ndiv_lt1:`

{1}	$\forall$ (a, b: int, c: posnat): $a < b \wedge \text{divides}(c, b) \supset \text{ndiv}(a, c) < \text{ndiv}(b, c)$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(a!1, c!1),
Adding type constraints for ndiv(b!1, c!1),
Case splitting on rem(c!1)(b!1) = 0,
we get 2 subgoals:

`both_sides_ndiv_lt1.1:`

{-1}	$\mathrm{rem}(c')(b') = 0$
{-2}	$b' = \mathrm{rem}(c')(b') + c' \times \mathrm{ndiv}(b', c')$
{-3}	$a' = \mathrm{rem}(c')(a') + c' \times \mathrm{ndiv}(a', c')$
{-4}	$\mathrm{integer\_pred}(a')$
{-5}	$\mathrm{integer\_pred}(b')$
{-6}	$c' > 0$
{-7}	$a' < b'$
{-8}	$\mathrm{divides}(c', b')$
{1}	$\mathrm{ndiv}(a', c') < \mathrm{ndiv}(b', c')$

Replacing using formula -1,

Hiding formulas: -1,

Simplifying, rewriting, and recording with decision procedures,

Applying both_sides_times_pos_ge2

Instantiating the top quantifier in -1 with the terms: c', $\mathrm{ndiv}(a', c')$, $\mathrm{ndiv}(b', c')$,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `both_sides_ndiv_lt1.1`.

`both_sides_ndiv_lt1.2:`

{-1}	$b' = \mathrm{rem}(c')(b') + c' \times \mathrm{ndiv}(b', c')$
{-2}	$a' = \mathrm{rem}(c')(a') + c' \times \mathrm{ndiv}(a', c')$
{-3}	$\mathrm{integer\_pred}(a')$
{-4}	$\mathrm{integer\_pred}(b')$
{-5}	$c' > 0$
{-6}	$a' < b'$
{-7}	$\mathrm{divides}(c', b')$
{1}	$\mathrm{rem}(c')(b') = 0$
{2}	$\mathrm{ndiv}(a', c') < \mathrm{ndiv}(b', c')$

Rewriting using rem_def2, matching in *,

This completes the proof of `both_sides_ndiv_lt1.2`.

Q.E.D.

C.143.19 Number_Props.rem_ndiv

Terse proof for `rem_ndiv`.

`rem_ndiv:`

{1}	$\forall$ (a: posnat, b: int, c: posnat): $\mathrm{rem}(a)(\mathrm{ndiv}(b, c)) = \mathrm{ndiv}(\mathrm{rem}(c \times a)(b), c)$

Repeatedly Skolemizing and flattening,

Rewriting using rem_prod_3, matching in *,

Applying ndiv_plus_mod

Instantiating the top quantifier in -1 with the terms: $\mathrm{rem}(a')(\mathrm{ndiv}(b', c')) \times c'$, c', $\mathrm{rem}(c')(b')$,

Rewriting using divides_prod2, matching in *,

we get 2 subgoals:

`rem_ndiv.1:`

{-1}	$\mathrm{ndiv}(\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c' + \mathrm{rem}(c')(b'),\ c') =$ $\mathrm{ndiv}(\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c',\ c')$
{-2}	$a' > 0$
{-3}	$\mathrm{integer\_pred}(b')$
{-4}	$c' > 0$
{1}	$\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) =$ $\mathrm{ndiv}(\mathrm{rem}(c')(b') + \mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c',\ c')$

Replacing using formula -1,
Hiding formulas: -1,
Rewriting using ndiv_times_divident_2, matching in *,
we get 2 subgoals:
`rem_ndiv.1.1:`

{-1}	$a' > 0$
{-2}	$\mathrm{integer\_pred}(b')$
{-3}	$c' > 0$
{1}	$\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) = \mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times \mathrm{ndiv}(c',\ c')$

Rewriting using ndiv_self, matching in *,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `rem_ndiv.1.1`.
`rem_ndiv.1.2:`

{-1}	$a' > 0$
{-2}	$\mathrm{integer\_pred}(b')$
{-3}	$c' > 0$
{1}	$\mathrm{divides}(c',\ c')$
{2}	$\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) = \mathrm{ndiv}(\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c',\ c')$

Rewriting using divides_reflexive, matching in *,
This completes the proof of `rem_ndiv.1.2`.
`rem_ndiv.2:`

{-1}	$\mathrm{divides}(c',\ \mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c') \supset$ $\mathrm{ndiv}(\mathrm{rem}(c')(b') + \mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c',\ c') =$ $\mathrm{ndiv}(\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c',\ c')$
{-2}	$a' > 0$
{-3}	$\mathrm{integer\_pred}(b')$
{-4}	$c' > 0$
{1}	$\mathrm{divides}(c',\ c')$
{2}	$\mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) =$ $\mathrm{ndiv}(\mathrm{rem}(c')(b') + \mathrm{rem}(a')(\mathrm{ndiv}(b',\ c')) \times c',\ c')$

Rewriting using divides_reflexive, matching in *,
This completes the proof of `rem_ndiv.2`.
Q.E.D.

C.143.20 Number_Props.mod_rem

Terse proof for `mod_rem`.
`mod_rem:`

{1}	$\forall\ (i\colon \mathrm{int},\ b\colon \mathrm{posnat})\colon \mathrm{mod}(i,\ b) = \mathrm{rem}(b)(i)$

Repeatedly Skolemizing and flattening,
Expanding the definition of mod,
Using lemma rem_floor,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `mod_rem`.
Q.E.D.

C.143.21 Number_Props.ndiv_rem_divisible_TCC1

Terse proof for `ndiv_rem_divisible_TCC1`.

`ndiv_rem_divisible_TCC1:`

{1}	$\forall\ (a,\ c\colon \text{posnat})\colon \text{divides}(c,\ a) \wedge c \leq a \supset \text{ndiv}(a,\ c) > 0$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(a!1, c!1),
Case splitting on rem(c!1)(a!1) = 0,
we get 2 subgoals:

`ndiv_rem_divisible_TCC1.1:`

{-1}	$\text{rem}(c')(a') = 0$
{-2}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a',\ c')$
{-3}	$\text{ndiv}(a',\ c') \leq a'$
{-4}	$a' > 0$
{-5}	$c' > 0$
{-6}	$\text{divides}(c',\ a')$
{-7}	$c' \leq a'$
{1}	$\text{ndiv}(a',\ c') > 0$

Replacing using formula -1,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_rem_divisible_TCC1.1`.

`ndiv_rem_divisible_TCC1.2:`

{-1}	$a' = \text{rem}(c')(a') + c' \times \text{ndiv}(a',\ c')$
{-2}	$\text{ndiv}(a',\ c') \leq a'$
{-3}	$a' > 0$
{-4}	$c' > 0$
{-5}	$\text{divides}(c',\ a')$
{-6}	$c' \leq a'$
{1}	$\text{rem}(c')(a') = 0$
{2}	$\text{ndiv}(a',\ c') > 0$

Rewriting using rem_def2, matching in *,
This completes the proof of `ndiv_rem_divisible_TCC1.2`.
Q.E.D.

C.143.22 Number_Props.ndiv_rem_divisible

Terse proof for `ndiv_rem_divisible`.

`ndiv_rem_divisible:`

{1}	$\forall$ (a, c: posnat, b: int): divides(c, a) $\wedge$ $c \leq a$ $\supset$ ndiv(rem(a)(b), c) = rem(ndiv(a, c))(ndiv(b, c))

Repeatedly Skolemizing and flattening,

Expanding the definition of divides,

Repeatedly Skolemizing and flattening,

Replacing using formula -5,

Rewriting using rem_ndiv, matching in *,

we get 2 subgoals:

`ndiv_rem_divisible.1:`

{-1}	integer_pred(x')
{-2}	$c' \times x' > 0$
{-3}	$c' > 0$
{-4}	integer_pred(b')
{-5}	$a' = c' \times x'$
{-6}	$c' \leq c' \times x'$
{1}	rem(x')(ndiv(b', c')) = rem(ndiv($c' \times x'$, c'))(ndiv(b', c'))

Rewriting using ndiv_times_divident_1, matching in *,

we get 2 subgoals:

`ndiv_rem_divisible.1.1:`

{-1}	integer_pred(x')
{-2}	$c' \times x' > 0$
{-3}	$c' > 0$
{-4}	integer_pred(b')
{-5}	$a' = c' \times x'$
{-6}	$c' \leq c' \times x'$
{1}	rem(x')(ndiv(b', c')) = rem(ndiv(c', c') $\times$ x')(ndiv(b', c'))

Rewriting using ndiv_self, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_rem_divisible.1.1`.

`ndiv_rem_divisible.1.2:`

{-1}	integer_pred(x')
{-2}	$c' \times x' > 0$
{-3}	$c' > 0$
{-4}	integer_pred(b')
{-5}	$a' = c' \times x'$
{-6}	$c' \leq c' \times x'$
{1}	divides(c', c')
{2}	rem(x')(ndiv(b', c')) = rem(ndiv($c' \times x'$, c'))(ndiv(b', c'))

Rewriting using divides_reflexive, matching in *,

This completes the proof of `ndiv_rem_divisible.1.2`.

`ndiv_rem_divisible.2:`

{-1} integer_pred(x')
{-2} $c' \times x' > 0$
{-3} $c' > 0$
{-4} integer_pred(b')
{-5} $a' = c' \times x'$
{-6} $c' \leq c' \times x'$

{1} $x' \geq 0 \wedge x' > 0$
{2} ndiv(rem($c' \times x'$)(b'), c') =
rem(ndiv($c' \times x'$, c'))(ndiv(b', c'))

Applying both_sides_times_pos_gt2
Instantiating the top quantifier in -1 with the terms: c', x', 0,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_rem_divisible.2`.
Q.E.D.

C.143.23 Number_Props.expt_2_8

Terse proof for `expt_2_8`.
`expt_2_8:`

{1} expt(2, 8) = 256

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expt_2_8`.
Q.E.D.

C.143.24 Number_Props.ndiv_0

Terse proof for `ndiv_0`.
`ndiv_0:`

{1} $\forall$ (b: posnat): ndiv(0, b) = 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ndiv_0`.
Q.E.D.

C.143.25 Number_Props.ndiv_1

Terse proof for `ndiv_1`.
`ndiv_1:`

{1} $\forall$ (i: int): ndiv(i, 1) = i

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(i!1, 1),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_1`.
Q.E.D.

C.143.26 Number_Props.ndiv_bigger

Terse proof for `ndiv_bigger`.
`ndiv_bigger:`

{1}	$\forall$ (n: nat, m: posnat): $n < m \supset \text{ndiv}(n, m) = 0$

Repeatedly Skolemizing and flattening,
Adding type constraints for ndiv(n!1, m!1),
Rewriting using rem_mod2, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Applying both_sides_times2
Instantiating the top quantifier in -1 with the terms: m', 0, ndiv(n', m'),
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_bigger`.
Q.E.D.

C.143.27 Number_Props.ndiv_reduce

Terse proof for `ndiv_reduce`.
`ndiv_reduce:`

{1}	$\forall$ (a: int, b: posnat, c: int, d: posnat): $a \times d = b \times c \supset \text{ndiv}(a, b) = \text{ndiv}(c, d)$

Repeatedly Skolemizing and flattening,
Rewriting using ndiv_floor, matching in *,
Rewriting using ndiv_floor, matching in *,
Case splitting on a!1 / b!1 = c!1 / d!1,
we get 2 subgoals:
`ndiv_reduce.1:`

{-1}	$a'/b' = c'/d'$
{-2}	integer_pred(a')
{-3}	$b' > 0$
{-4}	integer_pred(c')
{-5}	$d' > 0$
{-6}	$a' \times d' = b' \times c'$
{1}	floor(a'/b') = floor(c'/d')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_reduce.1`.
`ndiv_reduce.2:`

{-1}	integer_pred(a')
{-2}	$b' > 0$
{-3}	integer_pred(c')
{-4}	$d' > 0$
{-5}	$a' \times d' = b' \times c'$
{1}	$a'/b' = c'/d'$
{2}	floor(a'/b') = floor(c'/d')

Case splitting on a!1 / b!1 * b!1 = c!1 / d!1 * b!1,
we get 2 subgoals:

`ndiv_reduce.2.1:`

{-1} $a'/b' \times b' = c'/d' \times b'$
{-2} integer_pred(a')
{-3} $b' > 0$
{-4} integer_pred(c')
{-5} $d' > 0$
{-6} $a' \times d' = b' \times c'$

{1} $a'/b' = c'/d'$
{2} floor(a'/b') = floor(c'/d')

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_reduce.2.1`.
`ndiv_reduce.2.2:`

{-1} integer_pred(a')
{-2} $b' > 0$
{-3} integer_pred(c')
{-4} $d' > 0$
{-5} $a' \times d' = b' \times c'$

{1} $a'/b' \times b' = c'/d' \times b'$
{2} $a'/b' = c'/d'$
{3} floor(a'/b') = floor(c'/d')

Applying both_sides_times1
Instantiating the top quantifier in -1 with the terms: d', $a'/b' \times b'$, $c'/d' \times b'$,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_reduce.2.2`.
Q.E.D.

C.143.28 Number_Props.ndiv_expt_expt_TCC1

Terse proof for `ndiv_expt_expt_TCC1`.
`ndiv_expt_expt_TCC1:`

{1} $\forall$ (a: posnat, n, m: nat): $\neg\ n < m \wedge a > 1 \supset n - m \geq 0$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ndiv_expt_expt_TCC1`.
Q.E.D.

C.143.29 Number_Props.ndiv_expt_expt

Terse proof for `ndiv_expt_expt`.
`ndiv_expt_expt:`

{1} $\forall$ (a: posnat, n, m: nat):
$a > 1 \supset$
ndiv(expt(a, n), expt(a, m)) =
IF $n < m$ THEN 0 ELSE expt(a, $n - m$) ENDIF

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`ndiv_expt_expt.1:`

{-1}	$a' > 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$a' > 1$
{-5}	$n' < m'$
{1}	$\text{ndiv}(\text{expt}(a',\ n'),\ \text{expt}(a',\ m')) = 0$

Rewriting using ndiv_bigger, matching in *,
Applying both_sides_expt_gt1_lt_aux
Instantiating the top quantifier in -1 with the terms: a', $n' - 1$, $m' - 1$,
we get 2 subgoals:
`ndiv_expt_expt.1.1:`

{-1}	$\text{expt}(a',\ n' - 1 + 1) < \text{expt}(a',\ m' - 1 + 1) \equiv$ $n' - 1 < m' - 1$
{-2}	$a' > 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$a' > 1$
{-6}	$n' < m'$
{1}	$\text{expt}(a',\ n') < \text{expt}(a',\ m')$
{2}	$\text{ndiv}(\text{expt}(a',\ n'),\ \text{expt}(a',\ m')) = 0$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `ndiv_expt_expt.1.1`.
`ndiv_expt_expt.1.2:`

{-1}	$a' > 0$
{-2}	$n' \geq 0$
{-3}	$m' \geq 0$
{-4}	$a' > 1$
{-5}	$n' < m'$
{1}	$n' - 1 \geq 0$
{2}	$\text{expt}(a',\ n') < \text{expt}(a',\ m')$
{3}	$\text{ndiv}(\text{expt}(a',\ n'),\ \text{expt}(a',\ m')) = 0$

Expanding the definition of expt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Applying increasing_expt_aux
Instantiating the top quantifier in -1 with the terms: a', $m' - 2$,
we get 2 subgoals:
`ndiv_expt_expt.1.2.1:`

{-1}	$\text{expt}(a',\ m' - 2 + 2) > a'$
{-2}	$a' > 0$
{-3}	$n' \geq 0$
{-4}	$m' \geq 0$
{-5}	$a' > 1$
{-6}	$n' < m'$
{1}	$n' - 1 \geq 0$
{2}	$1 < \text{expt}(a',\ m')$
{3}	$\text{ndiv}(\text{expt}(a',\ n'),\ \text{expt}(a',\ m')) = 0$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_expt_expt.1.2.1`.

`ndiv_expt_expt.1.2.2`:

$$
\begin{array}{ll}
\{\text{-1}\} & a' > 0 \\
\{\text{-2}\} & n' \geq 0 \\
\{\text{-3}\} & m' \geq 0 \\
\{\text{-4}\} & a' > 1 \\
\{\text{-5}\} & n' < m' \\
\hline
\{1\} & m' - 2 \geq 0 \\
\{2\} & n' - 1 \geq 0 \\
\{3\} & 1 < \text{expt}(a', m') \\
\{4\} & \text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = 0
\end{array}
$$

Expanding the definition of expt,

Expanding the definition of expt,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `ndiv_expt_expt.1.2.2`.

`ndiv_expt_expt.2`:

$$
\begin{array}{ll}
\{\text{-1}\} & a' > 0 \\
\{\text{-2}\} & n' \geq 0 \\
\{\text{-3}\} & m' \geq 0 \\
\{\text{-4}\} & a' > 1 \\
\hline
\{1\} & n' < m' \\
\{2\} & \text{ndiv}(\text{expt}(a', n'), \text{expt}(a', m')) = \text{expt}(a', n' - m')
\end{array}
$$

Applying ndiv_reduce

Instantiating the top quantifier in -1 with the terms: expt(a', n'), expt(a', m'), expt(a', $n' - m'$), 1,

Rewriting using expt_plus_aux, matching in *,

Rewriting using ndiv_1, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `ndiv_expt_expt.2`.

Q.E.D.

C.143.30 Number_Props.min_plus1_TCC1

Terse proof for `min_plus1_TCC1`.

`min_plus1_TCC1`:

$$
\begin{array}{ll}
\{1\} & \forall\ (P\colon\ (\text{nonempty?}[\text{nat}])) : \\
 & \quad \neg\ P(0)\ \supset\ \text{nonempty?}[\text{nat}](\lambda\ (n\colon\ \text{nat})\colon\ P(n+1))
\end{array}
$$

Repeatedly Skolemizing and flattening,

Expanding the definition of nonempty?,

Expanding the definition of empty?,

Repeatedly Skolemizing and flattening,

Expanding the definition of member,

Case splitting on x!1 = 0,

we get 2 subgoals:

`min_plus1_TCC1.1:`

{-1}	$x' = 0$
{-2}	$x' \geq 0$
{-3}	$P'(x')$
{-4}	$\forall\ (x\colon\ \text{nat})\colon\ \neg\ P'(1+x)$
{1}	$P'(0)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `min_plus1_TCC1.1`.

`min_plus1_TCC1.2:`

{-1}	$x' \geq 0$
{-2}	$P'(x')$
{-3}	$\forall\ (x\colon\ \text{nat})\colon\ \neg\ P'(1+x)$
{1}	$x' = 0$
{2}	$P'(0)$

Instantiating the top quantifier in -3 with the terms: $x' - 1$,
we get 2 subgoals:

`min_plus1_TCC1.2.1:`

{-1}	$x' \geq 0$
{-2}	$P'(x')$
{1}	$x' = 0$
{2}	$P'(0)$
{3}	$P'(1 + x' - 1)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `min_plus1_TCC1.2.1`.

`min_plus1_TCC1.2.2:`

{-1}	$x' \geq 0$
{-2}	$P'(x')$
{1}	$x' - 1 \geq 0$
{2}	$x' = 0$
{3}	$P'(0)$

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `min_plus1_TCC1.2.2`.
Q.E.D.

C.143.31 Number_Props.min_plus1

Terse proof for `min_plus1`.

`min_plus1:`

{1}	$\forall\ (P\colon\ (\text{nonempty?}[\text{nat}]))\colon$ $\neg\ P(0)\ \supset\ \min(P)\ =\ \min(\lambda\ (n:\ \text{nat}):\ P(n+1)) + 1$

Repeatedly Skolemizing and flattening,
Using lemma min_def,
we get 3 subgoals:

`min_plus1.1:`

{-1} $\min(\lambda\ (n\colon\ \text{nat})\colon\ P'(n+1)) = \min(P') - 1 \equiv$
$\text{minimum?}(\min(P') - 1,\ \lambda\ (n\colon\ \text{nat})\colon\ P'(n+1))$
{-2} $\text{nonempty?}\lceil\text{nat}\rceil(P')$

{1} $P'(0)$
{2} $\min(P') = \min(\lambda\ (n:\ \text{nat}):\ P'(n+1)) + 1$

Simplifying, rewriting, and recording with decision procedures,
Using lemma min_def,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of minimum?,
Hiding formulas: 3,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `min_plus1.1`.

`min_plus1.2:`

{-1} $\text{nonempty?}\lceil\text{nat}\rceil(P')$

{1} $\text{nonempty?}\lceil\text{nat}\rceil(\lambda\ (n\colon\ \text{nat})\colon\ P'(1+n))$
{2} $P'(0)$
{3} $\min(P') = \min(\lambda\ (n:\ \text{nat}):\ P'(n+1)) + 1$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `min_plus1.2`.

`min_plus1.3:`

{-1} $\text{nonempty?}\lceil\text{nat}\rceil(P')$

{1} $\min\lceil\text{nat}\rceil(P') - 1 \geq 0$
{2} $P'(0)$
{3} $\min(P') = \min(\lambda\ (n:\ \text{nat}):\ P'(n+1)) + 1$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `min_plus1.3`.
Q.E.D.

C.143.32 Number_Props.min_member

Terse proof for `min_member`.

`min_member:`

{1} $\forall\ (P\colon\ (\text{nonempty?}\lceil\text{nat}\rceil),\ n\colon\ \text{nat})\colon\ \min(P) = n \supset P(n)$

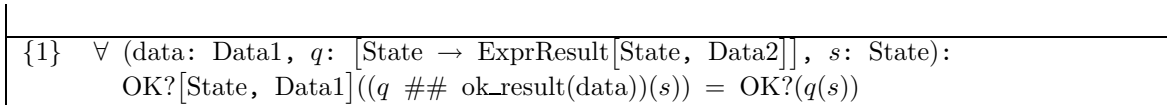
Repeatedly Skolemizing and flattening,
Rewriting using min_def, matching in *,
Expanding the definition of minimum?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `min_member`.
Q.E.D.

C.144 Proofs for Ok_Result_Rewrite (state-transformer.pvs)

C.144.1 Ok_Result_Rewrite.ok_result_q_ok

Terse proof for `ok_result_q_ok`.

`ok_result_q_ok`:

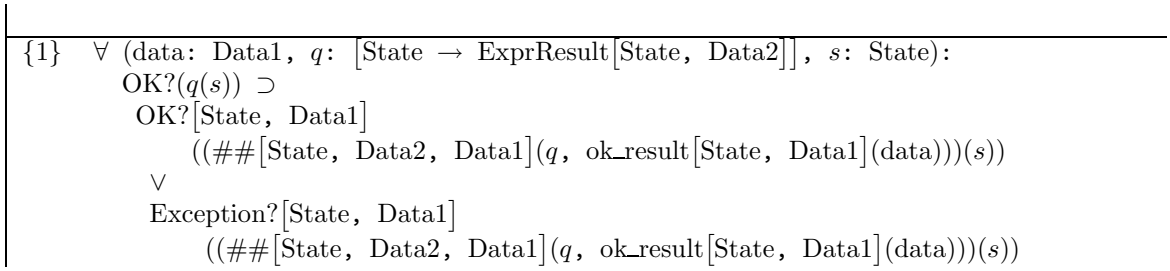


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_q_ok`.
Q.E.D.

C.144.2 Ok_Result_Rewrite.ok_result_q_state_TCC1

Terse proof for `ok_result_q_state_TCC1`.

`ok_result_q_state_TCC1`:

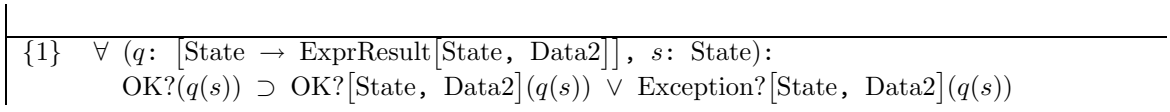


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_q_state_TCC1`.
Q.E.D.

C.144.3 Ok_Result_Rewrite.ok_result_q_state_TCC2

Terse proof for `ok_result_q_state_TCC2`.

`ok_result_q_state_TCC2`:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_q_state_TCC2`.
Q.E.D.

C.144.4 Ok_Result_Rewrite.ok_result_q_state

Terse proof for `ok_result_q_state`.

`ok_result_q_state`:

{1} ∀ (data: Data1, q: [State → ExprResult[State, Data2]], s: State):
OK?(q(s)) ⊃ state[State, Data1](((q ## ok_result(data))(s)) = state(q(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_q_state`.
Q.E.D.

C.144.5 Ok_Result_Rewrite.ok_result_q_data_TCC1

Terse proof for `ok_result_q_data_TCC1`.

`ok_result_q_data_TCC1`:

{1} ∀ (data: Data1, q: [State → ExprResult[State, Data2]], s: State):
OK?(q(s)) ⊃
OK?[State, Data1]
((##[State, Data2, Data1](q, ok_result[State, Data1](data)))(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_q_data_TCC1`.
Q.E.D.

C.144.6 Ok_Result_Rewrite.ok_result_q_data

Terse proof for `ok_result_q_data`.

`ok_result_q_data`:

{1} ∀ (data1: Data1, q: [State → ExprResult[State, Data2]], s: State):
OK?(q(s)) ⊃ data((q ## ok_result(data1))(s)) = data1

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_q_data`.
Q.E.D.

C.144.7 Ok_Result_Rewrite.ok_result_elemination_1

Terse proof for `ok_result_elemination_1`.

`ok_result_elemination_1`:

{1} ∀ (data: Data1, q: [State → ExprResult[State, Data2]]):
(ok_result[State, Data1](data) ## q) = q

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_elemination_1`.
Q.E.D.

C.145 Proofs for PDBR_Data_Model (paging-data-models.pvs)

C.145.1 PDBR_Data_Model.pdbr_valid?_TCC1

Terse proof for `pdbr_valid?_TCC1`.
`pdbr_valid?_TCC1:`

{1} ∀ (data: list[Byte]): length(data) = 4 ⊃ 1 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pdbr_valid?_TCC1`.
Q.E.D.

C.145.2 PDBR_Data_Model.pdbr_valid?_TCC2

Terse proof for `pdbr_valid?_TCC2`.
`pdbr_valid?_TCC2:`

{1} ∀ (data: list[Byte]):
cut_bits(nth(data, 1), 0, 4) = 0 ∧ length(data) = 4 ⊃ 0 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pdbr_valid?_TCC2`.
Q.E.D.

C.145.3 PDBR_Data_Model.pdbr_to_byte_TCC1

Terse proof for `pdbr_to_byte_TCC1`.
`pdbr_to_byte_TCC1:`

{1} ∀ (pdbr: Pdbr_type):
overwrite_bool_bit(overwrite_bool_bit(0, pdbr‘pwt, 3), pdbr‘pcd, 4) < max_byte

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pdbr_to_byte_TCC1`.
Q.E.D.

C.145.4 PDBR_Data_Model.pdbr_to_byte_TCC2

Terse proof for `pdbr_to_byte_TCC2`.
`pdbr_to_byte_TCC2:`

{1} ∀ (pdbr: Pdbr_type): offset(pdbr‘base_addr) ≥ 0

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pdbr_to_byte_TCC2`.
Q.E.D.

C.145.5 PDBR_Data_Model.pdbr_to_byte_TCC3

Terse proof for `pdbr_to_byte_TCC3`.

`pdbr_to_byte_TCC3:`

```
{1}  ∀ (pdbr: Pdbr_type):
       every[nat]
           ({s: nat | s < max_byte})
           ((:cut_bits(offset(pdbr`base_addr), 8, 8),
                cut_bits(offset(pdbr`base_addr), 16, 8),
                cut_bits(offset(pdbr`base_addr), 24, 8):))
```

Installing automatic rewrites from: every cut_bits_below max_byte min_bits_per_byte bits_per_byte_minimum

Repeatedly Skolemizing and flattening,

Using lemma bits_per_byte_minimum,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`pdbr_to_byte_TCC3.1:`

```
{-1}  bits_per_byte ≥ 8
{1}   cut_bits(offset(pdbr'`base_addr), 24, 8) < expt(2, bits_per_byte)
```

Using lemma cut_bits_below,

Using lemma both_sides_expt_gt1_le_aux,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pdbr_to_byte_TCC3.1`.

`pdbr_to_byte_TCC3.2:`

```
{-1}  bits_per_byte ≥ 8
{1}   cut_bits(offset(pdbr'`base_addr), 16, 8) < expt(2, bits_per_byte)
```

Using lemma cut_bits_below,

Using lemma both_sides_expt_gt1_le_aux,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pdbr_to_byte_TCC3.2`.

`pdbr_to_byte_TCC3.3:`

```
{-1}  bits_per_byte ≥ 8
{1}   cut_bits(offset(pdbr'`base_addr), 8, 8) < expt(2, bits_per_byte)
```

Using lemma cut_bits_below,

Using lemma both_sides_expt_gt1_le_aux,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pdbr_to_byte_TCC3.3`.

Q.E.D.

C.145.6 PDBR_Data_Model.pdbr_from_byte_TCC1

Terse proof for `pdbr_from_byte_TCC1`.

`pdbr_from_byte_TCC1:`

```
{1}  ∀ (data: list[Byte], a: Address): pdbr_valid?(data, a) ⊃ 1 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pdbr_from_byte_TCC1`.

Q.E.D.

C.145.7 PDBR_Data_Model.pdbr_from_byte_TCC2

Terse proof for `pdbr_from_byte_TCC2`.

`pdbr_from_byte_TCC2:`

{1} ∀ (data: list⌊Byte⌋, a: Address): pdbr_valid?(data, a) ⊃ 2 < length⌊Byte⌋(data)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pdbr_from_byte_TCC2`.

Q.E.D.

C.145.8 PDBR_Data_Model.pdbr_from_byte_TCC3

Terse proof for `pdbr_from_byte_TCC3`.

`pdbr_from_byte_TCC3:`

{1} ∀ (data: list⌊Byte⌋, a: Address): pdbr_valid?(data, a) ⊃ 3 < length⌊Byte⌋(data)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pdbr_from_byte_TCC3`.

Q.E.D.

C.145.9 PDBR_Data_Model.pdbr_from_byte_TCC4

Terse proof for `pdbr_from_byte_TCC4`.

pdbr_from_byte_TCC4:

```
{1}  ∀ (data: list[Byte], a: Address):
       pdbr_valid?(data, a) ⊃
        Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                                nth[Byte]
                                                                                (data, 1),
                                                                                8,
                                                                                8),
                                                                      nth[Byte](data, 2), 16, 8),
                                                            nth[Byte](data, 3), 24, 8))`type_of)
          ∧
          (0 ≤
             Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                                nth[Byte]
                                                                                (data, 1),
                                                                                8,
                                                                                8),
                                                                      nth[Byte](data, 2), 16, 8),
                                                            nth[Byte](data, 3), 24, 8))`offset
             ∧
             Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                                nth[Byte]
                                                                                (data, 1),
                                                                                8,
                                                                                8),
                                                                      nth[Byte](data, 2), 16, 8),
                                                            nth[Byte](data, 3), 24, 8))`offset
               < max_linear_offset)
          ∧
          aligned?(12)
                   (offset
                        (Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
                                                                       (0,
                                                                          nth[Byte](data, 1),
                                                                          8,
                                                                          8),
                                                                       nth[Byte](data, 2),
                                                                       16,
                                                                       8),
                                                             nth[Byte](data, 3), 24, 8))))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid? max_linear_offset bus_width

Expanding the definition of Mem,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using aligned_overwrite_shift_bits_more, matching in *,

Rewriting using aligned_overwrite_shift_bits_more, matching in *,

Rewriting using aligned_overwrite_shift_bits_less, matching in *,

Hiding formulas: 2, 3, 4,

Expanding the definition of pdbr_valid?,
Applying disjunctive simplification to flatten sequent,
Using lemma cut_bits_aligned,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pdbr_from_byte_TCC4`.
Q.E.D.

C.145.10 PDBR_Data_Model.pdbr_from_byte_TCC5

Terse proof for `pdbr_from_byte_TCC5`.
`pdbr_from_byte_TCC5:`

{1} ∀ (data: list[Byte], a: Address): pdbr_valid?(data, a) ⊃ 0 < length[Byte](data)

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid? max_linear_offset bus_width
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pdbr_valid?,
which is trivially true.
This completes the proof of `pdbr_from_byte_TCC5`.
Q.E.D.

C.145.11 PDBR_Data_Model.pdbr_model_TCC1

Terse proof for `pdbr_model_TCC1`.
`pdbr_model_TCC1:`

{1} ∀ (f: [[Pdbr_type, Address] → [list[Byte] → bool]]):
(f = (λ (d: Pdbr_type, a: Address): (zero_mask?(4)))) ⊃
(∀ (x_1: [Pdbr_type, Address]): singleton?[list[Byte]](f(x_1)))

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Expanding the definition of singleton?,
which is trivially true.
This completes the proof of `pdbr_model_TCC1`.
Q.E.D.

C.145.12 PDBR_Data_Model.pdbr_valid_to_byte

Terse proof for `pdbr_valid_to_byte`.
`pdbr_valid_to_byte:`

{1} ∀ (d: Pdbr_type, a: Address):
valid?(uidt(pdbr_model))(to_byte(pdbr_model)(d, a), a)

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: pdbr_model pdbr_to_byte pdbr_valid? nth length cut_bits_zero cut_bits_overwrite_bool_bit_disjoined cut_bits_overwrite_bits_disjoined cut_bits_cut_bits

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Adding type constraints for d!1‘base_addr,
Using lemma aligned_cut_bits,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pdbr_valid_to_byte`.
Q.E.D.

C.145.13 PDBR_Data_Model.pdbr_is_pod

Terse proof for `pdbr_is_pod`.
`pdbr_is_pod:`

{1} pod_data_type?(pdbr_model)

Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Expanding the definition of uninterpreted_data_type?,
Applying propositional simplification,
we get 6 subgoals:
`pdbr_is_pod.1:`

{1} ∀ (l: list[Byte], a: Address):
¬ length(l) = size(pdbr_model‘uidt) ⊃ ¬ valid?(pdbr_model‘uidt)(l, a)

Repeatedly Skolemizing and flattening,
Expanding the definition of pdbr_model,
Expanding the definition of pdbr_valid?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pdbr_is_pod.1`.
`pdbr_is_pod.2:`

{1} ∀ (d: Pdbr_type, a: Address):
valid?(uidt(pdbr_model))(to_byte(pdbr_model)(d, a), a)

Applying pdbr_valid_to_byte
which is trivially true.
This completes the proof of `pdbr_is_pod.2`.
`pdbr_is_pod.3:`

{1} ∀ (l: list[Byte], a: Address):
valid?(uidt(pdbr_model))(l, a) ≡ up?(from_byte(pdbr_model)(l, a))

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of pdbr_model,
Expanding the definition of pdbr_from_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pdbr_is_pod.3`.
`pdbr_is_pod.4:`

{1} ∀ (d: Pdbr_type, a: Address):
down(from_byte(pdbr_model)(to_byte(pdbr_model)(d, a), a)) = d

Repeatedly Skolemizing and flattening,

Using lemma pdbr_valid_to_byte,

Installing automatic rewrites from: pdbr_model pdbr_to_byte pdbr_from_byte pdbr_valid? nth length cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Adding type constraints for d!1‘base_addr,

Expanding the definition of max_linear_offset,

Expanding the definition of bus_width,

Rewriting using shift_bits_left_cut_bits, matching in *,

we get 2 subgoals:

`pdbr_is_pod.4.1:`

{-1}	Mem?(d'‘base_addr‘type_of)
{-2}	$0 \leq d'$‘base_addr‘offset
{-3}	d'‘base_addr‘offset $<$ expt(2, 32)
{-4}	aligned?(12)(offset(d'‘base_addr))
{-5}	cut_bits(offset(d'‘base_addr), 8, 4) $=$ 0
{1}	(#base_addr := Mem(offset(d'‘base_addr)), pcd := d'‘pcd, pwt := d'‘pwt#) $=$ d'

Applying extensionality,

Applying extensionality,

we get 2 subgoals:

`pdbr_is_pod.4.1.1:`

{-1}	Mem?(d'‘base_addr‘type_of)
{-2}	$0 \leq d'$‘base_addr‘offset
{-3}	d'‘base_addr‘offset $<$ expt(2, 32)
{-4}	aligned?(12)(offset(d'‘base_addr))
{-5}	cut_bits(offset(d'‘base_addr), 8, 4) $=$ 0
{1}	Mem(offset(d'‘base_addr))‘offset $=$ d'‘base_addr‘offset

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `pdbr_is_pod.4.1.1`.

`pdbr_is_pod.4.1.2:`

{-1}	Mem?(d'‘base_addr‘type_of)
{-2}	$0 \leq d'$‘base_addr‘offset
{-3}	d'‘base_addr‘offset $<$ expt(2, 32)
{-4}	aligned?(12)(offset(d'‘base_addr))
{-5}	cut_bits(offset(d'‘base_addr), 8, 4) $=$ 0
{1}	Mem(offset(d'‘base_addr))‘type_of $=$ d'‘base_addr‘type_of

Expanding the definition of Mem,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `pdbr_is_pod.4.1.2`.

pdbr_is_pod.4.2:

{-1} Mem?(d'`base_addr`type_of)
{-2} $0 \leq d'$`base_addr`offset
{-3} d'`base_addr`offset $<$ expt(2, 32)
{-4} aligned?(12)(offset(d'`base_addr))
{-5} cut_bits(offset(d'`base_addr), 8, 4) = 0

{1} aligned?(8)(offset(d'`base_addr))
{2} (#base_addr := Mem(shift_bits_left(cut_bits(offset(d'`base_addr), 8, 24), 8)),
pcd := d'`pcd,
pwt := d'`pwt#)
= d'

Applying aligned_bigger
Instantiating the top quantifier in -1 with the terms: offset(d'`base_addr), 12, 8,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of pdbr_is_pod.4.2.
pdbr_is_pod.5:

{1} ∀ (l: list[Byte], a_1, a_2: Address):
valid?(uidt(pdbr_model))(l, a_1) ≡ valid?(uidt(pdbr_model))(l, a_2)

Expanding the definition of pdbr_model,
Expanding the definition of pdbr_valid?,
which is trivially true.
This completes the proof of pdbr_is_pod.5.
pdbr_is_pod.6:

{1} size(uidt(pdbr_model)) > 0

Expanding the definition of pdbr_model,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pdbr_is_pod.6.
Q.E.D.

C.146 Proofs for PDBR_Datatype (paging-data.pvs)

C.146.1 PDBR_Datatype.pdbr_data_type_TCC1

Terse proof for pdbr_data_type_TCC1.
pdbr_data_type_TCC1:

{1} ∃ (x: (pod_data_type?[Pdbr_type])): TRUE

Using lemma pdbr_data_type_exists,
This completes the proof of pdbr_data_type_TCC1.
Q.E.D.

C.147 Proofs for PDBR_Type (paging-data.pvs)

C.147.1 PDBR_Type.Pdbr_type_TCC1

Terse proof for `Pdbr_type_TCC1`.
`Pdbr_type_TCC1:`

{1} ∀ (a: Memory_Address_4G): offset(a) ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Pdbr_type_TCC1`.
Q.E.D.

C.148 Proofs for PDE_Datatype (paging-data.pvs)

C.148.1 PDE_Datatype.pde_data_type_TCC1

Terse proof for `pde_data_type_TCC1`.
`pde_data_type_TCC1:`

{1} ∃ (x: (pod_data_type?⌊Pde_type⌋)): TRUE

Using lemma pde_data_type_exists,
This completes the proof of `pde_data_type_TCC1`.
Q.E.D.

C.149 Proofs for PDE_Model (paging-data-models.pvs)

C.149.1 PDE_Model.pde_valid?_TCC1

Terse proof for `pde_valid?_TCC1`.
`pde_valid?_TCC1:`

{1} ∀ (data: list⌊Byte⌋): length(data) = 4 ⊃ 0 < length⌊Byte⌋(data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_valid?_TCC1`.
Q.E.D.

C.149.2 PDE_Model.pde_valid?_TCC2

Terse proof for `pde_valid?_TCC2`.
`pde_valid?_TCC2:`

{1} ∀ (data: list⌊Byte⌋):
cut_bit(nth(data, 0), 7) ∧ cut_bit(nth(data, 0), 0) ∧ length(data) = 4 ⊃
1 < length⌊Byte⌋(data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_valid?_TCC2`.

Q.E.D.

C.149.3 PDE_Model.pde_valid?_TCC3

Terse proof for `pde_valid?_TCC3`.

`pde_valid?_TCC3:`

```
{1}  ∀ (data: list[Byte]):
       cut_bits(nth(data, 1), 5, 3) = 0 ∧
        cut_bit(nth(data, 0), 7) ∧ cut_bit(nth(data, 0), 0) ∧ length(data) = 4
        ⊃ 2 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pde_valid?_TCC3`.

Q.E.D.

C.149.4 PDE_Model.pde_pte_to_byte_TCC1

Terse proof for `pde_pte_to_byte_TCC1`.

`pde_pte_to_byte_TCC1:`

```
{1}  ∀ (pde: Pde_pt_type):
       overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit
                                                                  (overwrite_bool_bit
                                                                    (overwrite_bool_bit
                                                                      (0, TRUE, 0),
                                                                      mem-
     ory_access_to_bools
                                                                      (pde‘access)‘1,
                                                                      1),
                                                                    mem-
     ory_privilege_to_bool
                                                                      (pde‘privilege),
                                                                      2),
                                                                  pde‘write_through,
                                                                  3),
                                                    pde‘cache_disabled, 4),
                                       pde‘accessed, 5)
         < max_byte
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pde_pte_to_byte_TCC1`.

Q.E.D.

C.149.5 PDE_Model.pde_pte_to_byte_TCC2

Terse proof for `pde_pte_to_byte_TCC2`.

pde_pte_to_byte_TCC2:

```
{1}  ∀ (pde: Pde_pt_type): offset(pde‘page_table_base) ≥ 0
```

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_pte_to_byte_TCC2.

Q.E.D.

C.149.6 PDE_Model.pde_pte_to_byte_TCC3

Terse proof for pde_pte_to_byte_TCC3.

pde_pte_to_byte_TCC3:

```
{1}  ∀ (pde: Pde_pt_type):
       every[nat]
            ({s: nat | s < max_byte})
            ((:overwrite_shift_bits(overwrite_bool_bit(0, pde‘global_page, 0),
                                                cut_bits(offset(pde‘page_table_base), 12, 4), 4, 4),
                 cut_bits(offset(pde‘page_table_base), 16, 8),
                 cut_bits(offset(pde‘page_table_base), 24, 8):))
```

Installing automatic rewrites from: cut_bits_byte every

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_pte_to_byte_TCC3.

Q.E.D.

C.149.7 PDE_Model.pde_super_to_byte_TCC1

Terse proof for pde_super_to_byte_TCC1.

pde_super_to_byte_TCC1:

{1} ∀ (spage: Pde_4m_type):
overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit
(overwrite_bool_bit
(overwrite_bool_bit
(overwrite_bool_bit
(overwrite_bool_bit
(0, TRUE, 0),
memory_access_to_bools
(spage`access)`1,
1),
memory_privilege_to_bool
(spage`privilege),
2),
spage`write_through,
3),
spage`cache_disabled,
4),
spage`accessed,
5),
spage`dirty, 6),
TRUE, 7)
< max_byte

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less shift_bits_left_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_super_to_byte_TCC1.

Q.E.D.

C.149.8 PDE_Model.pde_super_to_byte_TCC2

Terse proof for pde_super_to_byte_TCC2.

pde_super_to_byte_TCC2:

{1} ∀ (spage: Pde_4m_type): offset(spage`page_base) ≥ 0

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_super_to_byte_TCC2.

Q.E.D.

C.149.9 PDE_Model.pde_super_to_byte_TCC3

Terse proof for pde_super_to_byte_TCC3.

pde_super_to_byte_TCC3:

```
{1}  ∀ (spage: Pde_4m_type):
       every[nat]
         ({s: nat | s < max_byte})
         ((:overwrite_bool_bit(overwrite_bool_bit(0, spage`global_page, 0),
                                 spage`page_table_attribute_index, 4),
             shift_bits_left(cut_bits(offset(spage`page_base), 22, 2), 6),
             cut_bits(offset(spage`page_base), 24, 8):))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less cut_bits_byte overwrite_shift_bits_less
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of shift_bits_left,
Using lemma cut_bits_below,
Expanding the definition of max_byte,
Using lemma bits_per_byte_minimum,
Expanding the definition of min_bits_per_byte,
Using lemma both_sides_expt_gt1_le_aux,
we get 2 subgoals:
pde_super_to_byte_TCC3.1:

```
{-1}  expt(2, 7 + 1) ≤ expt(2, bits_per_byte − 1 + 1) ≡
        7 ≤ bits_per_byte − 1
{-2}  bits_per_byte ≥ 8
{-3}  cut_bits(offset(spage'`page_base), 22, 2) < expt(2, 2)
-------------------------------------------------------------------------
{1}   cut_bits(offset(spage'`page_base), 22, 2) × expt(2, 6) < expt(2, bits_per_byte)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma expt_plus_aux,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_super_to_byte_TCC3.1.
pde_super_to_byte_TCC3.2:

```
{-1}  bits_per_byte ≥ 8
{-2}  cut_bits(offset(spage'`page_base), 22, 2) < expt(2, 2)
-------------------------------------------------------------------------
{1}   bits_per_byte − 1 ≥ 0
{2}   cut_bits(offset(spage'`page_base), 22, 2) × expt(2, 6) < expt(2, bits_per_byte)
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pde_super_to_byte_TCC3.2.
Q.E.D.

C.149.10 PDE_Model.pde_to_byte_TCC1

Terse proof for pde_to_byte_TCC1.
pde_to_byte_TCC1:

```
{1}  ∀ (pde: Pde_type): pde = Not_present ⊃ 0 < max_byte
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every less_than_max_byte expt_2_8

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pde_to_byte_TCC1`.
Q.E.D.

C.149.11 PDE_Model.pde_to_byte_TCC2

Terse proof for `pde_to_byte_TCC2`.
`pde_to_byte_TCC2`:

{1} ∀ (pde: Pde_type):
pde = Not_present ⊃
every[real]
(λ (x: real): rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
((:0, 0, 0:))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_to_byte_TCC2`.
Q.E.D.

C.149.12 PDE_Model.pde_4m_from_byte_TCC1

Terse proof for `pde_4m_from_byte_TCC1`.
`pde_4m_from_byte_TCC1`:

{1} ∀ (data: list[Byte], a: Address): pde_valid?(data, a) ⊃ 0 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_4m_from_byte_TCC1`.
Q.E.D.

C.149.13 PDE_Model.pde_4m_from_byte_TCC2

Terse proof for `pde_4m_from_byte_TCC2`.
`pde_4m_from_byte_TCC2`:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ cut_bit(nth(data, 0), 7) ⊃
((singleton [Memory_access](Read) ∪ {Execute}) ⊆ bools_to_memory_access(cut_bit(nth [Byte](data, 0),

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_4m_from_byte_TCC2`.
Q.E.D.

C.149.14 PDE_Model.pde_4m_from_byte_TCC3

Terse proof for `pde_4m_from_byte_TCC3`.
`pde_4m_from_byte_TCC3`:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ cut_bit(nth(data, 0), 7) ⊃
1 < length[Byte](data)

Repeatedly Skolemizing and flattening,

Expanding the definition of pde_valid?,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `pde_4m_from_byte_TCC3`.

Q.E.D.

C.149.15 PDE_Model.pde_4m_from_byte_TCC4

Terse proof for `pde_4m_from_byte_TCC4`.

`pde_4m_from_byte_TCC4`:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ cut_bit(nth(data, 0), 7) ⊃
2 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pde_4m_from_byte_TCC4`.

Q.E.D.

C.149.16 PDE_Model.pde_4m_from_byte_TCC5

Terse proof for `pde_4m_from_byte_TCC5`.

`pde_4m_from_byte_TCC5`:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ cut_bit(nth(data, 0), 7) ⊃
3 < length[Byte](data)

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid? bus_width max_linear_offset

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pde_4m_from_byte_TCC5`.

Q.E.D.

C.149.17 PDE_Model.pde_4m_from_byte_TCC6

Terse proof for `pde_4m_from_byte_TCC6`.

pde_4m_from_byte_TCC6:

```
{1}  ∀ (data: list[Byte], a: Address):
       pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ cut_bit(nth(data, 0), 7) ⊃
        Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                           cut_bits(nth[Byte](data, 2),
                                                                    6,
                                                                    2),
                                                           22, 2),
                                      nth[Byte](data, 3), 24, 8))`type_of)
         ∧
         (0 ≤
            Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                          cut_bits(nth[Byte](data, 2),
                                                                   6,
                                                                   2),
                                                          22, 2),
                                     nth[Byte](data, 3), 24, 8))`offset
            ∧
            Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                          cut_bits(nth[Byte](data, 2),
                                                                   6,
                                                                   2),
                                                          22, 2),
                                     nth[Byte](data, 3), 24, 8))`offset
             < max_linear_offset)
         ∧
         aligned?(22)
                  (offset
                       (Mem(overwrite_shift_bits(overwrite_shift_bits(0,
                                                                      cut_bits
                                                                      (nth[Byte](data, 2),
                                                                        6,
                                                                        2),
                                                                      22,
                                                                      2),
                                                 nth[Byte](data, 3), 24, 8))))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid? bus_width max_linear_offset

Expanding the definition of Mem,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of pde_4m_from_byte_TCC6.

Q.E.D.

C.149.18 PDE_Model.pde_4k_from_byte_TCC1

Terse proof for pde_4k_from_byte_TCC1.

pde_4k_from_byte_TCC1:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ ¬ cut_bit(nth(data, 0), 7) ⊃
((singleton [Memory_access](Read) ∪ {Execute}) ⊆ bools_to_memory_access(cut_bit(nth [Byte]

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC1.
Q.E.D.

C.149.19 PDE_Model.pde_4k_from_byte_TCC2

Terse proof for pde_4k_from_byte_TCC2.
pde_4k_from_byte_TCC2:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ ¬ cut_bit(nth(data, 0), 7) ⊃
1 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC2.
Q.E.D.

C.149.20 PDE_Model.pde_4k_from_byte_TCC3

Terse proof for pde_4k_from_byte_TCC3.
pde_4k_from_byte_TCC3:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ ¬ cut_bit(nth(data, 0), 7) ⊃
2 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC3.
Q.E.D.

C.149.21 PDE_Model.pde_4k_from_byte_TCC4

Terse proof for pde_4k_from_byte_TCC4.
pde_4k_from_byte_TCC4:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ ¬ cut_bit(nth(data, 0), 7) ⊃
3 < length[Byte](data)

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid? max_linear_offset bus_width
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pde_4k_from_byte_TCC4.

Q.E.D.

C.149.22 PDE_Model.pde_4k_from_byte_TCC5

Terse proof for `pde_4k_from_byte_TCC5`.

pde_4k_from_byte_TCC5:

{1} ∀ (data: list[Byte], a: Address):
pde_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ∧ ¬ cut_bit(nth(data, 0), 7) ⊃
Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
cut_
(nth
(
4
4)
12,
4),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))`type_of)
∧
(0 ≤
Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
cut_bi
(nth[B
(dat
4,
4),
12,
4),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))`offset
∧
Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
cut_bi
(nth[B
(dat
4,
4),
12,
4),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))`offset
< max_linear_offset)
∧
aligned?(12)
(offset
(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
(0,
cut_bits
(nth[Byte](data
4,
4),
12,
4),
nth[Byte](data, 2
16,
8),
nth[Byte](data, 3), 24, 8))))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pde_valid? bus_width max_linear_offset
Expanding the definition of Mem,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pde_4k_from_byte_TCC5`.
Q.E.D.

C.149.23 PDE_Model.pde_from_byte_TCC1

Terse proof for `pde_from_byte_TCC1`.
`pde_from_byte_TCC1`:

{1} ∀ (data: list[Byte], a: Address):
cut_bit(nth(data, 0), 7) ∧ cut_bit(nth(data, 0), 0) ∧ pde_valid?(data, a) ⊃
up?[Pde_4m_type](pde_4m_from_byte(data, a))

Repeatedly Skolemizing and flattening,
Expanding the definition of pde_4m_from_byte,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pde_from_byte_TCC1`.
Q.E.D.

C.149.24 PDE_Model.pde_from_byte_TCC2

Terse proof for `pde_from_byte_TCC2`.
`pde_from_byte_TCC2`:

{1} ∀ (data: list[Byte], a: Address):
¬ cut_bit(nth(data, 0), 7) ∧ cut_bit(nth(data, 0), 0) ∧ pde_valid?(data, a) ⊃
up?[Pde_pt_type](pde_4k_from_byte(data, a))

Repeatedly Skolemizing and flattening,
Expanding the definition of pde_4k_from_byte,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pde_from_byte_TCC2`.
Q.E.D.

C.149.25 PDE_Model.pde_model_TCC1

Terse proof for `pde_model_TCC1`.
`pde_model_TCC1`:

{1} ∀ (f: [[Pde_type, Address] → [list[Byte] → bool]]):
(f = (λ (d: Pde_type, a: Address): (zero_mask?(4)))) ⊃
(∀ (x1: [Pde_type, Address]): singleton?[list[Byte]](f(x1)))

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,

Expanding the definition of singleton?,
which is trivially true.
This completes the proof of `pde_model_TCC1`.
Q.E.D.

C.149.26 PDE_Model.pde_valid_to_byte

The LaTeX code for this proof is broken.

C.149.27 PDE_Model.pde_is_pod

Terse proof for `pde_is_pod`.
`pde_is_pod:`

{1} pod_data_type?(pde_model)

Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Expanding the definition of uninterpreted_data_type?,
Applying propositional simplification,
we get 6 subgoals:
`pde_is_pod.1:`

{1} ∀ (l: list⌊Byte⌋, a: Address):
¬ length(l) = size(pde_model`uidt) ⊃ ¬ valid?(pde_model`uidt)(l, a)

Repeatedly Skolemizing and flattening,
Expanding the definition of pde_model,
Expanding the definition of pde_valid?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `pde_is_pod.1`.
`pde_is_pod.2:`

{1} ∀ (d: Pde_type, a: Address): valid?(uidt(pde_model))(to_byte(pde_model)(d, a), a)

Repeatedly Skolemizing and flattening,
Rewriting using pde_valid_to_byte, matching in *,
This completes the proof of `pde_is_pod.2`.
`pde_is_pod.3:`

{1} ∀ (l: list⌊Byte⌋, a: Address):
valid?(uidt(pde_model))(l, a) ≡ up?(from_byte(pde_model)(l, a))

Repeatedly Skolemizing and flattening,
Expanding the definition of pde_model,
Expanding the definition of pde_from_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pde_is_pod.3`.

pde_is_pod.4:

{1} ∀ (d: Pde_type, a: Address):
down(from_byte(pde_model)(to_byte(pde_model)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Expanding the definition of pde_model,
Expanding the definition of pde_to_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 3 subgoals:
pde_is_pod.4.1:

{-1} not_present?(d′)
{1} down(pde_from_byte((:0, 0, 0, 0:), a′)) = d′

Installing automatic rewrites from: pde_from_byte! pde_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits cut_bit_overwrite_shift_bits length_to_byte from_byte_to_byte valid_to_byte cut_bit_zero
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pde_is_pod.4.1.
pde_is_pod.4.2:

{-1} pde_pt?(d′)
{1} down(pde_from_byte(pde_pte_to_byte(pde_pt(d′), a′), a′)) = d′

Installing automatic rewrites from: pde_pte_to_byte! pde_4k_from_byte pde_from_byte! pde_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits cut_bit_overwrite_shift_bits cut_bits_overwrite_shift_bits_contained length_to_byte from_byte_to_byte valid_to_byte cut_bit_zero bool_to_memory_access_iso bool_to_memory_privilege_iso max_linear_offset bus_width
Adding type constraints for pde_pt(d!1)‘page_table_base,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using shift_bits_left_cut_bits, matching in *,
Applying extensionality,
we get 3 subgoals:
pde_is_pod.4.2.1:

{-1} Mem?(pde_pt(d′)‘page_table_base‘type_of)
{-2} 0 ≤ pde_pt(d′)‘page_table_base‘offset
{-3} pde_pt(d′)‘page_table_base‘offset < expt(2, 32)
{-4} aligned?(12)(offset(pde_pt(d′)‘page_table_base))
{-5} pde_pt?(d′)
{1} (#page_table_base := Mem(offset(pde_pt(d′)‘page_table_base)),
global_page := pde_pt(d′)‘global_page,
accessed := pde_pt(d′)‘accessed,
cache_disabled := pde_pt(d′)‘cache_disabled,
write_through := pde_pt(d′)‘write_through,
privilege := pde_pt(d′)‘privilege,
access := bools_to_memory_access(memory_access_to_bools(pde_pt(d′)‘access)‘1, TRUE)#)
= pde_pt(d′)

Expanding the definition of bools_to_memory_access,
Expanding the definition of Mem,
Expanding the definition of memory_access_to_bools,
Applying extensionality,
we get 2 subgoals:
`pde_is_pod.4.2.1.1:`

```
{-1}  Mem?(pde_pt(d')`page_table_base`type_of)
{-2}  0 ≤ pde_pt(d')`page_table_base`offset
{-3}  pde_pt(d')`page_table_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pde_pt(d')`page_table_base))
{-5}  pde_pt?(d')
------------------------------------------------------------
{1}   IF (Write ∈ pde_pt(d')`access)
        THEN ((singleton(Read) ∪ {Write}) ∪ {Execute})
      ELSE (singleton(Read) ∪ {Execute})
      ENDIF
       = pde_pt(d')`access
```

Adding type constraints for pde_pt(d!1)`access,
Keeping (-1 1) and hiding *,
Applying extensionality,
Expanding the definition of subset?,
Instantiating the top quantifier in -1 with the terms: x',
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_is_pod.4.2.1.1`.
`pde_is_pod.4.2.1.2:`

```
{-1}  Mem?(pde_pt(d')`page_table_base`type_of)
{-2}  0 ≤ pde_pt(d')`page_table_base`offset
{-3}  pde_pt(d')`page_table_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pde_pt(d')`page_table_base))
{-5}  pde_pt?(d')
------------------------------------------------------------
{1}   (#type_of := Mem_, offset := offset(pde_pt(d')`page_table_base)#) =
       pde_pt(d')`page_table_base
```

Applying extensionality,
This completes the proof of `pde_is_pod.4.2.1.2`.
`pde_is_pod.4.2.2:`

```
{-1}  Mem?(pde_pt(d')`page_table_base`type_of)
{-2}  0 ≤ pde_pt(d')`page_table_base`offset
{-3}  pde_pt(d')`page_table_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pde_pt(d')`page_table_base))
{-5}  pde_pt?(d')
------------------------------------------------------------
{1}   Mem?(Mem(offset(pde_pt(d')`page_table_base))`type_of) ∧
       (0 ≤ Mem(offset(pde_pt(d')`page_table_base))`offset ∧
          Mem(offset(pde_pt(d')`page_table_base))`offset < expt(2, 32))
        ∧ aligned?(12)(offset(Mem(offset(pde_pt(d')`page_table_base))))
```

Expanding the definition of Mem,
which is trivially true.
This completes the proof of `pde_is_pod.4.2.2`.

`pde_is_pod.4.2.3:`

```
{-1}  Mem?(pde_pt(d')`page_table_base`type_of)
{-2}  0 ≤ pde_pt(d')`page_table_base`offset
{-3}  pde_pt(d')`page_table_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pde_pt(d')`page_table_base))
{-5}  pde_pt?(d')
------------------------------------------------------------------------------
{1}   ((singleton⌊Memory_access⌋(Read) ∪ {Execute}) ⊆ bools_to_memory_access(memory_access_to_bools(pde_pt(d'
```

Adding type constraints for pde_pt(d!1)`access,
Keeping (-1 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: x',
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pde_is_pod.4.2.3`.
`pde_is_pod.4.3:`

```
------------------------------------------------------------------------------
{1}   not_present?(d')
{2}   pde_pt?(d')
{3}   down(pde_from_byte(pde_super_to_byte(pde_4m(d'), a'), a')) = d'
```

Installing automatic rewrites from: pde_super_to_byte! pde_4m_from_byte pde_from_byte! pde_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits cut_bit_overwrite_shift_bits cut_bits_overwrite_shift_bits_contained cut_bits_overwrite_shift_bits_disjoint cut_bits_shift_bits_left_outside cut_bits_shift_bits_left_in length_to_byte from_byte_to_byte valid_to_byte cut_bit_zero bool_to_memory_access_iso bool_to_memory_privilege_iso max_linear_offset bus_width
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Adding type constraints for pde_4m(d!1)`page_base,
Rewriting using shift_bits_left_cut_bits, matching in *,
Applying extensionality,
we get 3 subgoals:
`pde_is_pod.4.3.1:`

```
{-1}  Mem?(pde_4m(d')`page_base`type_of)
{-2}  0 ≤ pde_4m(d')`page_base`offset
{-3}  pde_4m(d')`page_base`offset < expt(2, 32)
{-4}  aligned?(22)(offset(pde_4m(d')`page_base))
------------------------------------------------------------------------------
{1}   (#page_base := Mem(offset(pde_4m(d')`page_base)),
         page_table_attribute_index := pde_4m(d')`page_table_attribute_index,
         global_page := pde_4m(d')`global_page,
         dirty := pde_4m(d')`dirty,
         accessed := pde_4m(d')`accessed,
         cache_disabled := pde_4m(d')`cache_disabled,
         write_through := pde_4m(d')`write_through,
         privilege := pde_4m(d')`privilege,
         access := bools_to_memory_access(memory_access_to_bools(pde_4m(d')`access)`1, TRUE)#)
       = pde_4m(d')
{2}   not_present?(d')
{3}   pde_pt?(d')
```

Applying extensionality,

we get 2 subgoals:

`pde_is_pod.4.3.1.1:`

{-1}	Mem?(pde_4m(d')‘page_base‘type_of)
{-2}	$0 \leq$ pde_4m(d')‘page_base‘offset
{-3}	pde_4m(d')‘page_base‘offset $<$ expt(2, 32)
{-4}	aligned?(22)(offset(pde_4m(d')‘page_base))
{1}	bools_to_memory_access(memory_access_to_bools(pde_4m(d')‘access)‘1, TRUE) = pde_4m(d')‘access
{2}	not_present?(d')
{3}	pde_pt?(d')

Expanding the definition of bools_to_memory_access,

Expanding the definition of memory_access_to_bools,

Adding type constraints for pde_4m(d!1)‘access,

Applying extensionality,

Expanding the definition of subset?,

Instantiating the top quantifier in -1 with the terms: x',

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pde_is_pod.4.3.1.1`.

`pde_is_pod.4.3.1.2:`

{-1}	Mem?(pde_4m(d')‘page_base‘type_of)
{-2}	$0 \leq$ pde_4m(d')‘page_base‘offset
{-3}	pde_4m(d')‘page_base‘offset $<$ expt(2, 32)
{-4}	aligned?(22)(offset(pde_4m(d')‘page_base))
{1}	Mem(offset(pde_4m(d')‘page_base)) = pde_4m(d')‘page_base
{2}	not_present?(d')
{3}	pde_pt?(d')

Expanding the definition of Mem,

Applying extensionality,

This completes the proof of `pde_is_pod.4.3.1.2`.

`pde_is_pod.4.3.2:`

{-1}	Mem?(pde_4m(d')‘page_base‘type_of)
{-2}	$0 \leq$ pde_4m(d')‘page_base‘offset
{-3}	pde_4m(d')‘page_base‘offset $<$ expt(2, 32)
{-4}	aligned?(22)(offset(pde_4m(d')‘page_base))
{1}	Mem?(Mem(offset(pde_4m(d')‘page_base))‘type_of) $\wedge$ ($0 \leq$ Mem(offset(pde_4m(d')‘page_base))‘offset $\wedge$ Mem(offset(pde_4m(d')‘page_base))‘offset $<$ expt(2, 32)) $\wedge$ aligned?(22)(offset(Mem(offset(pde_4m(d')‘page_base)))))
{2}	not_present?(d')
{3}	pde_pt?(d')

Expanding the definition of Mem,

which is trivially true.

This completes the proof of `pde_is_pod.4.3.2`.

pde_is_pod.4.3.3:

```
{-1}  Mem?(pde_4m(d')`page_base`type_of)
{-2}  0 ≤ pde_4m(d')`page_base`offset
{-3}  pde_4m(d')`page_base`offset < expt(2, 32)
{-4}  aligned?(22)(offset(pde_4m(d')`page_base))
------------------------------------------------------------------------------
{1}   ((singleton[Memory_access](Read) ∪ {Execute}) ⊆ bools_to_memory_access(memory_access_to_bools(pde_4m(d
{2}   not_present?(d')
{3}   pde_pt?(d')
```

Keeping (-1 1) and hiding *,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pde_is_pod.4.3.3.
pde_is_pod.5:

```
------------------------------------------------------------------------------
{1}   ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(pde_model))(l, a1) ≡ valid?(uidt(pde_model))(l, a2)
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of pde_model,
Expanding the definition of pde_valid?,
which is trivially true.
This completes the proof of pde_is_pod.5.
pde_is_pod.6:

```
------------------------------------------------------------------------------
{1}   size(uidt(pde_model)) > 0
```

Expanding the definition of pde_model,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pde_is_pod.6.
Q.E.D.

C.150 Proofs for PF_EC_Model (paging-data-models.pvs)

C.150.1 PF_EC_Model.pf_ec_valid?_TCC1

Terse proof for pf_ec_valid?_TCC1.
pf_ec_valid?_TCC1:

```
------------------------------------------------------------------------------
{1}   ∀ (data: list[Byte]): length(data) = 4 ⊃ 3 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pf_ec_valid?_TCC1.
Q.E.D.

C.150.2 PF_EC_Model.pf_ec_valid?_TCC2

Terse proof for `pf_ec_valid?_TCC2`.

`pf_ec_valid?_TCC2`:

```
{1}  ∀ (data: list[Byte]):
       cut_bits(nth(data, 3), 0, 8) = 0 ∧ length(data) = 4 ⊃ 2 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pf_ec_valid?_TCC2`.

Q.E.D.

C.150.3 PF_EC_Model.pf_ec_valid?_TCC3

Terse proof for `pf_ec_valid?_TCC3`.

`pf_ec_valid?_TCC3`:

```
{1}  ∀ (data: list[Byte]):
       cut_bits(nth(data, 2), 0, 8) = 0 ∧
        cut_bits(nth(data, 3), 0, 8) = 0 ∧ length(data) = 4
        ⊃ 1 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pf_ec_valid?_TCC3`.

Q.E.D.

C.150.4 PF_EC_Model.pf_ec_valid?_TCC4

Terse proof for `pf_ec_valid?_TCC4`.

`pf_ec_valid?_TCC4`:

```
{1}  ∀ (data: list[Byte]):
       cut_bits(nth(data, 1), 0, 8) = 0 ∧
        cut_bits(nth(data, 2), 0, 8) = 0 ∧
         cut_bits(nth(data, 3), 0, 8) = 0 ∧ length(data) = 4
        ⊃ 0 < length[Byte](data)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pf_ec_valid?_TCC4`.

Q.E.D.

C.150.5 PF_EC_Model.pf_ec_to_byte_TCC1

Terse proof for `pf_ec_to_byte_TCC1`.

pf_ec_to_byte_TCC1:

{1} ∀ (pf_ec: PF_error_code_type):
overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit
(overwrite_bool_bit
(0, pf_ec‘present, 0),
(Write ∈ pf_ec‘access),
1),
memory_privilege_to_bool
(pf_ec‘privilege),
2),
pf_ec‘reserved_bit_violation, 3),
(Execute ∈ pf_ec‘access), 4)
< max_byte

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pf_ec_to_byte_TCC1.
Q.E.D.

C.150.6 PF_EC_Model.pf_ec_to_byte_TCC2

Terse proof for pf_ec_to_byte_TCC2.

pf_ec_to_byte_TCC2:

{1} ∀ (pf_ec: PF_error_code_type):
every[real]
(λ (x: real): rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
((:0, 0, 0:))

Expanding the definition of max_byte,
Adding type constraints for expt(2, bits_per_byte),
Installing automatic rewrites from: every
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pf_ec_to_byte_TCC2.
Q.E.D.

C.150.7 PF_EC_Model.pf_ec_from_byte_TCC1

Terse proof for pf_ec_from_byte_TCC1.

pf_ec_from_byte_TCC1:

{1} ∀ (data: list[Byte], a: Address): pf_ec_valid?(data, a) ⊃ 0 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pf_ec_from_byte_TCC1.
Q.E.D.

C.150.8 PF_EC_Model.pf_ec_from_byte_TCC2

Terse proof for pf_ec_from_byte_TCC2.

pf_ec_from_byte_TCC2:

{1} ∀ (data: list[Byte], a: Address):
pf_ec_valid?(data, a) ⊃
(Read ∈ bools_to_memory_access(cut_bit(nth[Byte](data, 0), 1), cut_bit(nth [Byte](data, 0)

Repeatedly Skolemizing and flattening,
Expanding the definition of every,
Expanding the definition of bools_to_memory_access,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pf_ec_from_byte_TCC2.
Q.E.D.

C.150.9 PF_EC_Model.pf_ec_model_TCC1

Terse proof for pf_ec_model_TCC1.

pf_ec_model_TCC1:

{1} ∀ (f: [[PF_error_code_type, Address] → [list[Byte] → bool]]):
(f = (λ (d: PF_error_code_type, a: Address): (zero_mask?(4)))) ⊃
(∀ (x1: [PF_error_code_type, Address]): singleton?[list[Byte]](f(x1)))

Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Expanding the definition of singleton?,
which is trivially true.
This completes the proof of pf_ec_model_TCC1.
Q.E.D.

C.150.10 PF_EC_Model.pf_ec_valid_to_byte

Terse proof for pf_ec_valid_to_byte.

pf_ec_valid_to_byte:

{1} ∀ (d: PF_error_code_type, a: Address):
valid?(uidt(pf_ec_model))(to_byte(pf_ec_model)(d, a), a)

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: pf_ec_model pf_ec_to_byte pf_ec_valid? nth length cut_bits_zero cut_bits_overwrite_bool_bit_disjoined cut_bits_overwrite_bits_disjoined cut_bits_cut_bits
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pf_ec_valid_to_byte.
Q.E.D.

C.150.11 PF_EC_Model.pf_ec_is_pod

Terse proof for pf_ec_is_pod.

pf_ec_is_pod:

{1} pod_data_type?(pf_ec_model)

Expanding the definition of pod_data_type?,
Expanding the definition of interpreted_data_type?,
Expanding the definition of uninterpreted_data_type?,
Applying propositional simplification,
we get 6 subgoals:
pf_ec_is_pod.1:

{1} ∀ (l: list[Byte], a: Address):
¬ length(l) = size(pf_ec_model‘uidt) ⊃ ¬ valid?(pf_ec_model‘uidt)(l, a)

Repeatedly Skolemizing and flattening,
Expanding the definition of pf_ec_model,
Expanding the definition of pf_ec_valid?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of pf_ec_is_pod.1.
pf_ec_is_pod.2:

{1} ∀ (d: PF_error_code_type, a: Address):
valid?(uidt(pf_ec_model))(to_byte(pf_ec_model)(d, a), a)

Repeatedly Skolemizing and flattening,
Rewriting using pf_ec_valid_to_byte, matching in *,
This completes the proof of pf_ec_is_pod.2.
pf_ec_is_pod.3:

{1} ∀ (l: list[Byte], a: Address):
valid?(uidt(pf_ec_model))(l, a) ≡ up?(from_byte(pf_ec_model)(l, a))

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of pf_ec_model,
Expanding the definition of pf_ec_from_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pf_ec_is_pod.3.
pf_ec_is_pod.4:

{1} ∀ (d: PF_error_code_type, a: Address):
down(from_byte(pf_ec_model)(to_byte(pf_ec_model)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Using lemma pf_ec_valid_to_byte,
Installing automatic rewrites from: pf_ec_model pf_ec_to_byte pf_ec_from_byte pf_ec_valid? nth length cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit bool_to_memory_access_iso bool_to_memory_privilege_iso
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Applying decompose-equality,
Expanding the definition of bools_to_memory_access,
Keeping (1) and hiding *,

Adding type constraints for d!1‘access,
Applying extensionality,
Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pf_ec_is_pod.4`.
`pf_ec_is_pod.5`:

{1} ∀ (l: list[Byte], a_1, a_2: Address):
valid?(uidt(pf_ec_model))(l, a_1) ≡ valid?(uidt(pf_ec_model))(l, a_2)

Expanding the definition of pf_ec_model,
Expanding the definition of pf_ec_valid?,
which is trivially true.
This completes the proof of `pf_ec_is_pod.5`.
`pf_ec_is_pod.6`:

{1} size(uidt(pf_ec_model)) > 0

Expanding the definition of pf_ec_model,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pf_ec_is_pod.6`.
Q.E.D.

C.151 Proofs for PF_Error_Code_Type (paging-data.pvs)

C.151.1 PF_Error_Code_Type.pf_ec_data_type_TCC1

Terse proof for `pf_ec_data_type_TCC1`.
`pf_ec_data_type_TCC1`:

{1} ∃ (x: (pod_data_type?[PF_error_code_type])): TRUE

Using lemma pf_ec_type_exists,
This completes the proof of `pf_ec_data_type_TCC1`.
Q.E.D.

C.152 Proofs for PTE_Datatype (paging-data.pvs)

C.152.1 PTE_Datatype.pte_data_type_TCC1

Terse proof for `pte_data_type_TCC1`.
`pte_data_type_TCC1`:

{1} ∃ (x: (pod_data_type?[Pte_type])): TRUE

Using lemma pte_data_type_exists,
This completes the proof of `pte_data_type_TCC1`.
Q.E.D.

C.153 Proofs for PTE_Model (paging-data-models.pvs)

C.153.1 PTE_Model.pte_page_to_byte_TCC1

Terse proof for `pte_page_to_byte_TCC1`.

`pte_page_to_byte_TCC1:`

```
{1}  ∀ (pte: Pte_4k_type):
       overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit(overwrite_bool_bit
                                                                (overwrite_bool_bit
                                                                  (overwrite_bool_bit
                                                                    (overwrite_bool_bit
                                                                      (overwrite_bool_bit
                                                                        (0, TRUE, 0),
                                                                        mem-
     ory_access_to_bools
                                                                        (pte‘access)‘1,
                                                                        1),
                                                                      mem-
     ory_privilege_to_bool
                                                                        (pte‘privilege),
                                                                        2),
                                                                      pte‘write_through,
                                                                      3),
                                                                    pte‘cache_disabled,
                                                                    4),
                                                                  pte‘accessed,
                                                                  5),
                                                    pte‘dirty, 6),
                                      pte‘page_table_attribute_index, 7)
         < max_byte
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte overwrite_bool_bit_less overwrite_shift_bits_less max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_page_to_byte_TCC1`.

Q.E.D.

C.153.2 PTE_Model.pte_page_to_byte_TCC2

Terse proof for `pte_page_to_byte_TCC2`.

`pte_page_to_byte_TCC2:`

```
{1}  ∀ (pte: Pte_4k_type): offset(pte‘page_base) ≥ 0
```

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_page_to_byte_TCC2`.

Q.E.D.

C.153.3 PTE_Model.pte_page_to_byte_TCC3

Terse proof for `pte_page_to_byte_TCC3`.

`pte_page_to_byte_TCC3:`

```
{1}   ∀ (pte: Pte_4k_type):
        every[nat]
            ({s: nat | s < max_byte})
            ((:overwrite_shift_bits(overwrite_bool_bit(0, pte`global_page, 0),
                                    cut_bits(offset(pte`page_base), 12, 4), 4, 4),
                cut_bits(offset(pte`page_base), 16, 8),
                cut_bits(offset(pte`page_base), 24, 8):))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte cut_bits_byte overwrite_bool_bit_less overwrite_shift_bits_less

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_page_to_byte_TCC3`.

Q.E.D.

C.153.4 PTE_Model.pte_to_byte_TCC1

Terse proof for `pte_to_byte_TCC1`.

`pte_to_byte_TCC1:`

```
{1}   ∀ (pte: Pte_type): pte = Not_present ⊃ 0 < max_byte
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every less_than_max_byte expt_2_8

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_to_byte_TCC1`.

Q.E.D.

C.153.5 PTE_Model.pte_to_byte_TCC2

Terse proof for `pte_to_byte_TCC2`.

`pte_to_byte_TCC2:`

```
{1}   ∀ (pte: Pte_type):
        pte = Not_present ⊃
         every[real]
            (λ (x: real): rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            ((:0, 0, 0:))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of max_byte,

Adding type constraints for expt(2, bits_per_byte),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_to_byte_TCC2`.

Q.E.D.

C.153.6 PTE_Model.pte_page_from_byte_TCC1

Terse proof for `pte_page_from_byte_TCC1`.

`pte_page_from_byte_TCC1`:

{1} ∀ (data: list[Byte], a: Address): pte_valid?(data, a) ⊃ 0 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_page_from_byte_TCC1`.
Q.E.D.

C.153.7 PTE_Model.pte_page_from_byte_TCC2

Terse proof for `pte_page_from_byte_TCC2`.

`pte_page_from_byte_TCC2`:

{1} ∀ (data: list[Byte], a: Address):
pte_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ⊃
((singleton [Memory_access](Read) ∪ {Execute}) ⊆ bools_to_memory_access(cut_bit(nth [Byte](data, 0),

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_page_from_byte_TCC2`.
Q.E.D.

C.153.8 PTE_Model.pte_page_from_byte_TCC3

Terse proof for `pte_page_from_byte_TCC3`.

`pte_page_from_byte_TCC3`:

{1} ∀ (data: list[Byte], a: Address):
pte_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ⊃ 1 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_page_from_byte_TCC3`.
Q.E.D.

C.153.9 PTE_Model.pte_page_from_byte_TCC4

Terse proof for `pte_page_from_byte_TCC4`.

`pte_page_from_byte_TCC4`:

{1} ∀ (data: list[Byte], a: Address):
pte_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ⊃ 2 < length[Byte](data)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_page_from_byte_TCC4`.
Q.E.D.

C.153.10 PTE_Model.pte_page_from_byte_TCC5

Terse proof for `pte_page_from_byte_TCC5`.

`pte_page_from_byte_TCC5:`

{1} ∀ (data: list[Byte], a: Address): pte_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ⊃ 3 < length[Byte](data)

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: every aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pte_valid? max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `pte_page_from_byte_TCC5`.

Q.E.D.

C.153.11 PTE_Model.pte_page_from_byte_TCC6

Terse proof for `pte_page_from_byte_TCC6`.

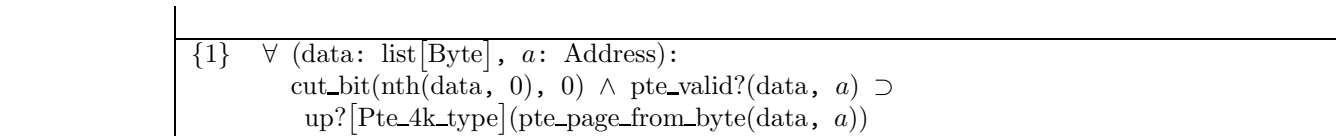
pte_page_from_byte_TCC6:

{1} ∀ (data: list[Byte], a: Address):
pte_valid?(data, a) ∧ cut_bit(nth(data, 0), 0) ⊃
Mem?(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
cut_bits
(nth[Byte]
(data, 1),
4,
4),
12,
4),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))`type_of)
∧
(0 ≤
Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
cut_bits
(nth[Byte]
(data, 1),
4,
4),
12,
4),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))`offset
∧
Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits(0,
cut_bits
(nth[Byte]
(data, 1),
4,
4),
12,
4),
nth[Byte](data, 2), 16, 8),
nth[Byte](data, 3), 24, 8))`offset
< max_linear_offset)
∧
aligned?(12)
(offset
(Mem(overwrite_shift_bits(overwrite_shift_bits(overwrite_shift_bits
(0,
cut_bits
(nth[Byte](data, 1),
4,
4),
12,
4),
nth[Byte](data, 2),
16,
8),
nth[Byte](data, 3), 24, 8))))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: every aligned_address? aligned_overwrite_shift_bits_more aligned_overwrite_shift_bits_less aligned_zero max_linear_expt_val overwrite_shift_bits_less pte_valid? max_linear_offset bus_width
Expanding the definition of Mem,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pte_page_from_byte_TCC6`.
Q.E.D.

C.153.12 PTE_Model.pte_from_byte_TCC1

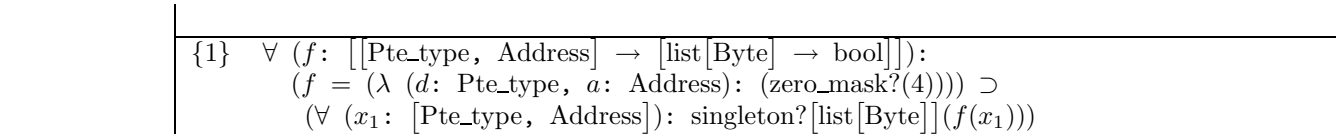
Terse proof for `pte_from_byte_TCC1`.
`pte_from_byte_TCC1:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_from_byte_TCC1`.
Q.E.D.

C.153.13 PTE_Model.pte_model_TCC1

Terse proof for `pte_model_TCC1`.
`pte_model_TCC1:`



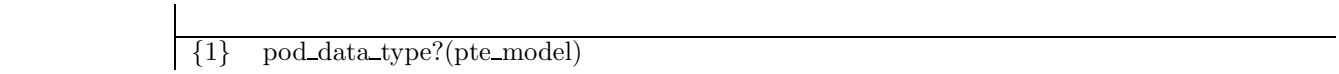
Repeatedly Skolemizing and flattening,
Replacing using formula -1,
Using lemma singleton_zero_mask,
Expanding the definition of singleton?,
which is trivially true.
This completes the proof of `pte_model_TCC1`.
Q.E.D.

C.153.14 PTE_Model.pte_valid_to_byte

The LaTeX code for this proof is broken.

C.153.15 PTE_Model.pte_is_pod

Terse proof for `pte_is_pod`.
`pte_is_pod:`



Expanding the definition of pod_data_type?,

Expanding the definition of interpreted_data_type?,
Expanding the definition of uninterpreted_data_type?,
Applying propositional simplification,
we get 6 subgoals:
`pte_is_pod.1:`

{1} ∀ (l: list[Byte], a: Address):
¬ length(l) = size(pte_model`uidt) ⊃ ¬ valid?(pte_model`uidt)(l, a)

Repeatedly Skolemizing and flattening,
Expanding the definition of pte_model,
Expanding the definition of pte_valid?,
which is trivially true.
This completes the proof of `pte_is_pod.1`.
`pte_is_pod.2:`

{1} ∀ (d: Pte_type, a: Address): valid?(uidt(pte_model))(to_byte(pte_model)(d, a), a)

Repeatedly Skolemizing and flattening,
Rewriting using pte_valid_to_byte, matching in *,
This completes the proof of `pte_is_pod.2`.
`pte_is_pod.3:`

{1} ∀ (l: list[Byte], a: Address):
valid?(uidt(pte_model))(l, a) ≡ up?(from_byte(pte_model)(l, a))

Repeatedly Skolemizing and flattening,
Expanding the definition of pte_model,
Expanding the definition of pte_from_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pte_is_pod.3`.
`pte_is_pod.4:`

{1} ∀ (d: Pte_type, a: Address):
down(from_byte(pte_model)(to_byte(pte_model)(d, a), a)) = d

Repeatedly Skolemizing and flattening,
Expanding the definition of pte_model,
Expanding the definition of pte_to_byte,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`pte_is_pod.4.1:`

{-1} not_present?(d')
{1} down(pte_from_byte((:0, 0, 0, 0:), a')) = d'

Installing automatic rewrites from: pte_from_byte cut_bit_zero nth pte_valid? length pte_page_from_byte
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pte_is_pod.4.1`.

`pte_is_pod.4.2:`

{1}	not_present?(d')
{2}	down(pte_from_byte(pte_page_to_byte(pte(d'), a'), a')) = d'

Installing automatic rewrites from: pte_page_to_byte! pte_from_byte pte_page_from_byte pte_valid? nth length length_cdr length_is_cons cut_bits_overwrite_bool_bit_disjoined cut_bits_zero cut_bits_overwrite_bits_disjoined cut_bits_cut_bits cut_bits_overwrite_bits_contained cut_bit_overwrite_bool_bit overwrite_shift_bits_zero overwrite_shift_bits_merge aligned_address_aligned cut_bit_cut_bits cut_bit_overwrite_shift_bits cut_bits_overwrite_shift_bits_contained length_to_byte from_byte_to_byte valid_to_byte cut_bit_zero bool_to_memory_access_iso bool_to_memory_privilege_iso max_linear_offset bus_width

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Adding type constraints for pte(d!1)‘page_base,
Rewriting using shift_bits_left_cut_bits, matching in *,
Applying extensionality,
we get 3 subgoals:
`pte_is_pod.4.2.1:`

{-1}	Mem?(pte(d')‘page_base‘type_of)
{-2}	0 ≤ pte(d')‘page_base‘offset
{-3}	pte(d')‘page_base‘offset < expt(2, 32)
{-4}	aligned?(12)(offset(pte(d')‘page_base))
{1}	(#page_base := Mem(offset(pte(d')‘page_base)), global_page := pte(d')‘global_page, page_table_attribute_index := pte(d')‘page_table_attribute_index, dirty := pte(d')‘dirty, accessed := pte(d')‘accessed, cache_disabled := pte(d')‘cache_disabled, write_through := pte(d')‘write_through, privilege := pte(d')‘privilege, access := bools_to_memory_access(memory_access_to_bools(pte(d')‘access)‘1, TRUE)#) = pte(d')
{2}	not_present?(d')

Applying extensionality,
we get 2 subgoals:
`pte_is_pod.4.2.1.1:`

{-1}	Mem?(pte(d')‘page_base‘type_of)
{-2}	0 ≤ pte(d')‘page_base‘offset
{-3}	pte(d')‘page_base‘offset < expt(2, 32)
{-4}	aligned?(12)(offset(pte(d')‘page_base))
{1}	bools_to_memory_access(memory_access_to_bools(pte(d')‘access)‘1, TRUE) = pte(d')‘access
{2}	not_present?(d')

Applying extensionality,
Expanding the definition of bools_to_memory_access,
Expanding the definition of memory_access_to_bools,
Adding type constraints for pte(d!1)‘access,
Expanding the definition of subset?,
Instantiating the top quantifier in -1 with the terms: x',
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_is_pod.4.2.1.1`.

`pte_is_pod.4.2.1.2:`

```
{-1}  Mem?(pte(d')`page_base`type_of)
{-2}  0 ≤ pte(d')`page_base`offset
{-3}  pte(d')`page_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pte(d')`page_base))
------------------------------------------------------------------------
{1}   Mem(offset(pte(d')`page_base)) = pte(d')`page_base
{2}   not_present?(d')
```

Expanding the definition of Mem,
Applying extensionality,
This completes the proof of `pte_is_pod.4.2.1.2`.

`pte_is_pod.4.2.2:`

```
{-1}  Mem?(pte(d')`page_base`type_of)
{-2}  0 ≤ pte(d')`page_base`offset
{-3}  pte(d')`page_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pte(d')`page_base))
------------------------------------------------------------------------
{1}   Mem?(Mem(offset(pte(d')`page_base))`type_of) ∧
        (0 ≤ Mem(offset(pte(d')`page_base))`offset ∧
           Mem(offset(pte(d')`page_base))`offset < expt(2, 32))
          ∧ aligned?(12)(offset(Mem(offset(pte(d')`page_base))))
{2}   not_present?(d')
```

Expanding the definition of Mem,
which is trivially true.
This completes the proof of `pte_is_pod.4.2.2`.

`pte_is_pod.4.2.3:`

```
{-1}  Mem?(pte(d')`page_base`type_of)
{-2}  0 ≤ pte(d')`page_base`offset
{-3}  pte(d')`page_base`offset < expt(2, 32)
{-4}  aligned?(12)(offset(pte(d')`page_base))
------------------------------------------------------------------------
{1}   ((singleton[Memory_access](Read) ∪ {Execute}) ⊆ bools_to_memory_access(memory_access_to_bools(pte(d')`ac
{2}   not_present?(d')
```

Expanding the definition of bools_to_memory_access,
Expanding the definition of memory_access_to_bools,
Adding type constraints for pte(d!1)`access,
Expanding the definition of subset?,
Repeatedly Skolemizing and flattening,
Instantiating the top quantifier in -1 with the terms: x',
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pte_is_pod.4.2.3`.

`pte_is_pod.5:`

```
------------------------------------------------------------------------
{1}   ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(pte_model))(l, a1) ≡ valid?(uidt(pte_model))(l, a2)
```

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Expanding the definition of pte_model,
Expanding the definition of pte_valid?,
which is trivially true.
This completes the proof of `pte_is_pod.5`.

pte_is_pod.6:

{1} size(uidt(pte_model)) > 0

Expanding the definition of pte_model,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pte_is_pod.6.
Q.E.D.

C.154 Proofs for Page_Directory_Types (paging-data.pvs)

C.154.1 Page_Directory_Types.Pde_pt_type_TCC1

Terse proof for Pde_pt_type_TCC1.
Pde_pt_type_TCC1:

{1} ∀ (a: Memory_Address_4G): offset(a) ≥ 0

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of Pde_pt_type_TCC1.
Q.E.D.

C.155 Proofs for Paging_Datatype (paging-data.pvs)

C.155.1 Paging_Datatype.paging_data_type_TCC1

Terse proof for paging_data_type_TCC1.
paging_data_type_TCC1:

{1} ∀ (lvl: Level): lvl = 0 ⊃ (∀ (x: ((range_pt(lvl)))): (pdir?)(x))

Repeatedly Skolemizing and flattening,
Expanding the definition of range_pt,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of paging_data_type_TCC1.
Q.E.D.

C.155.2 Paging_Datatype.paging_data_type_TCC2

Terse proof for paging_data_type_TCC2.
paging_data_type_TCC2:

{1} ∀ (lvl: Level): lvl = 0 ⊃ (∀ (x_1: Pde_type): range_pt(lvl)(Pdir(x_1)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of paging_data_type_TCC2.
Q.E.D.

C.155.3 Paging_Datatype.paging_data_type_TCC3

Terse proof for `paging_data_type_TCC3`.

`paging_data_type_TCC3:`

```
{1}  ∀ (lvl: Level):
       lvl = 0 ⊃
         pod_data_type?[(range_pt(lvl))]
           (dt_lift[Pde_type, (range_pt(lvl))]
             (pde_data_type, restrict[(pdir?), (range_pt(lvl)), Pde_type](pdir), Pdir))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of dt_lift,

Adding type constraints for pde_data_type,

Expanding the definition of pod_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of interpreted_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`paging_data_type_TCC3.1:`

```
{-1}  uninterpreted_data_type?(pde_data_type`uidt)
{-2}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-4}  ∀ (d: Pde_type, a: Address):
        down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(pde_data_type))(l, a_1) ≡ valid?(uidt(pde_data_type))(l, a_2)
{-6}  size(uidt(pde_data_type)) > 0
{-7}  lvl′ < max_level
{-8}  lvl′ = 0
-------------------------------------------------------------------------
{1}  ∀ (d: ((range_pt(lvl′))), a_1: Address):
       down(CASES from_byte(pde_data_type)
                    (to_byte(pde_data_type)
                       (restrict[(pdir?), ((range_pt(lvl′))), Pde_type](pdir)(d),
                        a_1),
                     a_1) OF
              up(d_1): up(Pdir(d_1)), bottom: bottom
            ENDCASES)
        = d
```

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`paging_data_type_TCC3.1.1:`

```
{-1}  range_pt(lvl′)(d′)
{-2}  uninterpreted_data_type?(pde_data_type`uidt)
{-3}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-4}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-5}  ∀ (d: Pde_type, a: Address):
        down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d
{-6}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(pde_data_type))(l, a1) ≡ valid?(uidt(pde_data_type))(l, a2)
{-7}  size(uidt(pde_data_type)) > 0
{-8}  lvl′ < max_level
{-9}  lvl′ = 0
{-10} up?(from_byte(pde_data_type)
                   (to_byte(pde_data_type)
                            (restrict[(pdir?), ((range_pt(lvl′))), Pde_type](pdir)(d′), a′),
                    a′))
------------------------------------------------------------------------------
{1}   Pdir(down(from_byte(pde_data_type)
                          (to_byte(pde_data_type)
                                   (restrict[(pdir?), ((range_pt(lvl′))), Pde_type](pdir)(d′),
                                    a′),
                           a′)))
        = d′
```

Instantiating the top quantifier in -5 with the terms: (pdir(d!1) a!1),
we get 2 subgoals:

`paging_data_type_TCC3.1.1.1:`

```
{-1}  range_pt(lvl′)(d′)
{-2}  uninterpreted_data_type?(pde_data_type`uidt)
{-3}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-4}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-5}  down(from_byte(pde_data_type)(to_byte(pde_data_type)(pdir(d′), a′), a′)) =
        pdir(d′)
{-6}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(pde_data_type))(l, a1) ≡ valid?(uidt(pde_data_type))(l, a2)
{-7}  size(uidt(pde_data_type)) > 0
{-8}  lvl′ < max_level
{-9}  lvl′ = 0
{-10} up?(from_byte(pde_data_type)
                   (to_byte(pde_data_type)
                            (restrict[(pdir?), ((range_pt(lvl′))), Pde_type](pdir)(d′), a′),
                    a′))
------------------------------------------------------------------------------
{1}   Pdir(down(from_byte(pde_data_type)
                          (to_byte(pde_data_type)
                                   (restrict[(pdir?), ((range_pt(lvl′))), Pde_type](pdir)(d′),
                                    a′),
                           a′)))
        = d′
```

Expanding the definition of restrict,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -5,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma Paging_type_Pdir_eta,
This completes the proof of `paging_data_type_TCC3.1.1.1`.
`paging_data_type_TCC3.1.1.2:`

```
{-1}  range_pt(lvl')(d')
{-2}  uninterpreted_data_type?(pde_data_type`uidt)
{-3}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-4}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-5}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(pde_data_type))(l, a1) ≡ valid?(uidt(pde_data_type))(l, a2)
{-6}  size(uidt(pde_data_type)) > 0
{-7}  lvl' < max_level
{-8}  lvl' = 0
{-9}  up?(from_byte(pde_data_type)
                   (to_byte(pde_data_type)
                            (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d'), a'),
                    a'))
|-------
{1}   pdir?(d')
{2}   Pdir(down(from_byte(pde_data_type)
                          (to_byte(pde_data_type)
                                   (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d'),
                                    a'),
                           a')))
      = d'
```

Expanding the definition of range_pt,
which is trivially true.
This completes the proof of `paging_data_type_TCC3.1.1.2`.
`paging_data_type_TCC3.1.2:`

```
{-1}  range_pt(lvl')(d')
{-2}  uninterpreted_data_type?(pde_data_type`uidt)
{-3}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-4}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-5}  ∀ (d: Pde_type, a: Address):
        down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d
{-6}  ∀ (l: list[Byte], a1, a2: Address):
        valid?(uidt(pde_data_type))(l, a1) ≡ valid?(uidt(pde_data_type))(l, a2)
{-7}  size(uidt(pde_data_type)) > 0
{-8}  lvl' < max_level
{-9}  lvl' = 0
|-------
{1}   up?(from_byte(pde_data_type)
                   (to_byte(pde_data_type)
                            (restrict[(pdir?), ((range_pt(lvl'))), Pde_type](pdir)(d'), a'),
                    a'))
{2}   down(bottom) = d'
```

Expanding the definition of restrict,
Instantiating quantified variables,

Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `paging_data_type_TCC3.1.2`.
`paging_data_type_TCC3.2`:

```
{-1}  uninterpreted_data_type?(pde_data_type`uidt)
{-2}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-4}  ∀ (d: Pde_type, a: Address):
        down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(pde_data_type))(l, a_1) ≡ valid?(uidt(pde_data_type))(l, a_2)
{-6}  size(uidt(pde_data_type)) > 0
{-7}  lvl′ < max_level
{-8}  lvl′ = 0
-------------------------------------------------------------------------------
{1}   ∀ (l_1: list[Byte], a_1: Address):
        valid?(uidt(pde_data_type))(l_1, a_1) ≡
          up?(CASES from_byte(pde_data_type)(l_1, a_1) OF up(d_1): up(Pdir(d_1)), bot-
      tom: bottom
                ENDCASES)
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `paging_data_type_TCC3.2`.
`paging_data_type_TCC3.3`:

```
{-1}  uninterpreted_data_type?(pde_data_type`uidt)
{-2}  ∀ (d: Pde_type, a: Address):
        valid?(uidt(pde_data_type))(to_byte(pde_data_type)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pde_data_type))(l, a) ≡ up?(from_byte(pde_data_type)(l, a))
{-4}  ∀ (d: Pde_type, a: Address):
        down(from_byte(pde_data_type)(to_byte(pde_data_type)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(pde_data_type))(l, a_1) ≡ valid?(uidt(pde_data_type))(l, a_2)
{-6}  size(uidt(pde_data_type)) > 0
{-7}  lvl′ < max_level
{-8}  lvl′ = 0
-------------------------------------------------------------------------------
{1}   ∀ (d: ((range_pt(lvl′))), a_1: Address):
        valid?(uidt(pde_data_type))
            (to_byte(pde_data_type)
                  (restrict[(pdir?), ((range_pt(lvl′))), Pde_type](pdir)(d), a_1),
              a_1)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of restrict,
Instantiating quantified variables,
This completes the proof of `paging_data_type_TCC3.3`.
Q.E.D.

C.155.4 Paging_Datatype.paging_data_type_TCC4

Terse proof for `paging_data_type_TCC4`.

`paging_data_type_TCC4:`

{1} ∀ (lvl: Level): lvl = 1 ⊃ (∀ (x: ((range_pt(lvl)))): (ptab?)(x))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `paging_data_type_TCC4`.

Q.E.D.

C.155.5 Paging_Datatype.paging_data_type_TCC5

Terse proof for `paging_data_type_TCC5`.

`paging_data_type_TCC5:`

{1} ∀ (lvl: Level): lvl = 1 ⊃ (∀ (x_1: Pte_type): range_pt(lvl)(Ptab(x_1)))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `paging_data_type_TCC5`.

Q.E.D.

C.155.6 Paging_Datatype.paging_data_type_TCC6

Terse proof for `paging_data_type_TCC6`.

`paging_data_type_TCC6:`

{1} ∀ (lvl: Level):
lvl = 1 ⊃
pod_data_type?[(range_pt(lvl))]
(dt_lift[Pte_type, (range_pt(lvl))]
(pte_data_type, restrict[(ptab?), (range_pt(lvl)), Pte_type](ptab), Ptab))

Expanding the definition of restrict,

Expanding the definition of dt_lift,

Repeatedly Skolemizing and flattening,

Adding type constraints for pte_data_type,

Expanding the definition of pod_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of interpreted_data_type?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`paging_data_type_TCC6.1:`

```
{-1}  uninterpreted_data_type?(pte_data_type`uidt)
{-2}  ∀ (d: Pte_type, a: Address):
        valid?(uidt(pte_data_type))(to_byte(pte_data_type)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pte_data_type))(l, a) ≡ up?(from_byte(pte_data_type)(l, a))
{-4}  ∀ (d: Pte_type, a: Address):
        down(from_byte(pte_data_type)(to_byte(pte_data_type)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(pte_data_type))(l, a_1) ≡ valid?(uidt(pte_data_type))(l, a_2)
{-6}  size(uidt(pte_data_type)) > 0
{-7}  lvl′ < max_level
{-8}  lvl′ = 1
-------------------------------------------------------------------------------
{1}   ∀ (d: ((range_pt(lvl′))), a_1: Address):
        down(CASES from_byte(pte_data_type)(to_byte(pte_data_type)(ptab(d), a_1), a_1) OF
               up(d_1): up(Ptab(d_1)), bottom: bottom
               ENDCASES)
          = d
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Instantiating quantified variables,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Replacing using formula -5,

Using lemma Paging_type_Ptab_eta,

This completes the proof of `paging_data_type_TCC6.1`.

`paging_data_type_TCC6.2:`

```
{-1}  uninterpreted_data_type?(pte_data_type`uidt)
{-2}  ∀ (d: Pte_type, a: Address):
        valid?(uidt(pte_data_type))(to_byte(pte_data_type)(d, a), a)
{-3}  ∀ (l: list[Byte], a: Address):
        valid?(uidt(pte_data_type))(l, a) ≡ up?(from_byte(pte_data_type)(l, a))
{-4}  ∀ (d: Pte_type, a: Address):
        down(from_byte(pte_data_type)(to_byte(pte_data_type)(d, a), a)) = d
{-5}  ∀ (l: list[Byte], a_1, a_2: Address):
        valid?(uidt(pte_data_type))(l, a_1) ≡ valid?(uidt(pte_data_type))(l, a_2)
{-6}  size(uidt(pte_data_type)) > 0
{-7}  lvl′ < max_level
{-8}  lvl′ = 1
-------------------------------------------------------------------------------
{1}   ∀ (l_1: list[Byte], a_1: Address):
        valid?(uidt(pte_data_type))(l_1, a_1) ≡
         up?(CASES from_byte(pte_data_type)(l_1, a_1) OF up(d_1): up(Ptab(d_1)), bot-
      tom: bottom
               ENDCASES)
```

Repeatedly Skolemizing and flattening,

Instantiating quantified variables,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `paging_data_type_TCC6.2`.

paging_data_type_TCC6.3:

{-1}	uninterpreted_data_type?(pte_data_type`uidt)
{-2}	∀ (d: Pte_type, a: Address): valid?(uidt(pte_data_type))(to_byte(pte_data_type)(d, a), a)
{-3}	∀ (l: list[Byte], a: Address): valid?(uidt(pte_data_type))(l, a) ≡ up?(from_byte(pte_data_type)(l, a))
{-4}	∀ (d: Pte_type, a: Address): down(from_byte(pte_data_type)(to_byte(pte_data_type)(d, a), a)) = d
{-5}	∀ (l: list[Byte], a_1, a_2: Address): valid?(uidt(pte_data_type))(l, a_1) ≡ valid?(uidt(pte_data_type))(l, a_2)
{-6}	size(uidt(pte_data_type)) > 0
{-7}	lvl′ < max_level
{-8}	lvl′ = 1
{1}	∀ (d: ((range_pt(lvl′))), a_1: Address): valid?(uidt(pte_data_type))(to_byte(pte_data_type)(ptab(d), a_1), a_1)

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
This completes the proof of paging_data_type_TCC6.3.
Q.E.D.

C.155.7 Paging_Datatype.paging_data_type_TCC7

Terse proof for paging_data_type_TCC7.

paging_data_type_TCC7:

{1}	∀ (lvl: Level): lvl = 0 ∨ lvl = 1

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of paging_data_type_TCC7.
Q.E.D.

C.156 Proofs for Paging_type (paging-data.pvs)

This theory contains no provable formal statements.

C.157 Proofs for Paging_type_adt (Paging_type_adt.pvs)

This theory contains no provable formal statements.

C.158 Proofs for Paging_type_adt_reduce (Paging_type_adt.pvs)

This theory contains no provable formal statements.

C.159 Proofs for Paging_type_helpers (paging-data.pvs)

C.159.1 Paging_type_helpers.range_pt_TCC1

Terse proof for `range_pt_TCC1`.

`range_pt_TCC1`:

{1} ∀ (lvl: Level): lvl = 0 ∨ lvl = 1

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `range_pt_TCC1`.
Q.E.D.

C.159.2 Paging_type_helpers.base_TCC1

Terse proof for `base_TCC1`.

`base_TCC1`:

{1} ∀ (pt: (present?), ptab: Pte_type): pt = Ptab(ptab) ⊃ pte?(ptab)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `base_TCC1`.
Q.E.D.

C.159.3 Paging_type_helpers.base_TCC2

Terse proof for `base_TCC2`.

`base_TCC2`:

{1} ∀ (pt: (present?), pdir: Pde_type): pt = Pdir(pdir) ⊃ ¬ not_present?(pdir)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `base_TCC2`.
Q.E.D.

C.159.4 Paging_type_helpers.set_reference_range

Terse proof for `set_reference_range`.

`set_reference_range`:

{1} ∀ (lvl: Level, pt: (range_pt(lvl)), access: Memory_access):
range_pt(lvl)(set_reference(pt, access))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `set_reference_range`.
Q.E.D.

C.159.5 Paging_type_helpers.set_reference_pdir_entry

Terse proof for `set_reference_pdir_entry`.

`set_reference_pdir_entry:`

```
{1}  ∀ (pt1, pt2: Paging_type, access: Memory_access):
       set_reference(pt1, access) = set_reference(pt2, access) ⊃
         pdir_entry?(pt1) = pdir_entry?(pt2)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `set_reference_pdir_entry`.
Q.E.D.

C.159.6 Paging_type_helpers.set_reference_ptab_entry

Terse proof for `set_reference_ptab_entry`.
`set_reference_ptab_entry:`

```
{1}  ∀ (pt1, pt2: Paging_type, access: Memory_access):
       set_reference(pt1, access) = set_reference(pt2, access) ⊃
         ptab_entry?(pt1) = ptab_entry?(pt2)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `set_reference_ptab_entry`.
Q.E.D.

C.159.7 Paging_type_helpers.set_reference_paging_type

Terse proof for `set_reference_paging_type`.
`set_reference_paging_type:`

```
{1}  ∀ (pt1, pt2: Paging_type, access: Memory_access, lvl: Level):
       set_reference(pt1, access) = set_reference(pt2, access) ⊃
         paging_type?(lvl, pt1) = paging_type?(lvl, pt2)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of paging_type?,
Case splitting on lvl!1 = 0,
we get 2 subgoals:
`set_reference_paging_type.1:`

```
{-1}  lvl′ = 0
{-2}  lvl′ < max_level
{-3}  set_reference(pt1′, access′) = set_reference(pt2′, access′)
{1}   COND lvl′ = 0 → pdir_entry?(pt1′), ELSE → ptab_entry?(pt1′) ENDCOND =
        COND lvl′ = 0 → pdir_entry?(pt2′), ELSE → ptab_entry?(pt2′) ENDCOND
```

Using lemma set_reference_pdir_entry,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `set_reference_paging_type.1`.
`set_reference_paging_type.2:`

```
{-1}  lvl′ < max_level
{-2}  set_reference(pt1′, access′) = set_reference(pt2′, access′)
{1}   lvl′ = 0
{2}   COND lvl′ = 0 → pdir_entry?(pt1′), ELSE → ptab_entry?(pt1′) ENDCOND =
        COND lvl′ = 0 → pdir_entry?(pt2′), ELSE → ptab_entry?(pt2′) ENDCOND
```

Using lemma set_reference_ptab_entry,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `set_reference_paging_type.2`.
Q.E.D.

C.159.8 Paging_type_helpers.set_reference_present

Terse proof for `set_reference_present`.

`set_reference_present:`

```
{1}  ∀ (pt1, pt2: Paging_type, access: Memory_access):
       set_reference(pt1, access) = set_reference(pt2, access) ⊃ present?(pt1) = present?(pt2)
```

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
we get 12 subgoals:

`set_reference_present.1:`

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-4}  ptab?(pt2′)
{-5}  not_present?(ptab(pt2′))
------------------------------------------------------------------------------
 {1}  not_present?(ptab(pt1′))
```

Applying decompose-equality,
This completes the proof of `set_reference_present.1`.

`set_reference_present.2:`

```
{-1}  Write?(access′)
{-2}  pde_pt?(pdir(pt1′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}  not_present?(pdir(pt2′))
------------------------------------------------------------------------------
 {1}  ptab?(pt1′)
 {2}  ptab?(pt2′)
```

Applying decompose-equality,
This completes the proof of `set_reference_present.2`.

`set_reference_present.3:`

```
{-1}  Write?(access′)
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Not_present)
{-3}  not_present?(pdir(pt2′))
------------------------------------------------------------------------------
 {1}  ptab?(pt1′)
 {2}  not_present?(pdir(pt1′))
 {3}  pde_pt?(pdir(pt1′))
 {4}  ptab?(pt2′)
```

Applying decompose-equality,
This completes the proof of `set_reference_present.3`.

`set_reference_present.4:`

```
{-1}  ptab?(pt1′)
{-2}  Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) = Ptab(Not_present)
{-3}  ptab?(pt2′)
{-4}  not_present?(ptab(pt2′))
------------------------------------------------------------------------
{1}   Write?(access′)
{2}   not_present?(ptab(pt1′))
```

Applying decompose-equality,
This completes the proof of `set_reference_present.4`.
`set_reference_present.5:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}  not_present?(pdir(pt2′))
------------------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
```

Applying decompose-equality,
This completes the proof of `set_reference_present.5`.
`set_reference_present.6:`

```
{-1}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-2}  not_present?(pdir(pt2′))
------------------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   not_present?(pdir(pt1′))
{4}   pde_pt?(pdir(pt1′))
{5}   ptab?(pt2′)
```

Applying decompose-equality,
This completes the proof of `set_reference_present.6`.
`set_reference_present.7:`

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  not_present?(ptab(pt1′))
{-4}  Ptab(Not_present) = Ptab(Pte(pte(ptab(pt2′)) WITH [dirty := TRUE, ac-
      cessed := TRUE]))
{-5}  ptab?(pt2′)
------------------------------------------------------------------------
{1}   not_present?(ptab(pt2′))
```

Applying decompose-equality,
This completes the proof of `set_reference_present.7`.
`set_reference_present.8:`

```
{-1}  Write?(access′)
{-2}  not_present?(pdir(pt1′))
{-3}  pde_pt?(pdir(pt2′))
{-4}  Pdir(Not_present) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   ptab?(pt2′)
```

Applying decompose-equality,
This completes the proof of `set_reference_present.8`.

`set_reference_present.9`:

{-1}	Write?(access′)
{-2}	not_present?(pdir(pt1′))
{-3}	Pdir(Not_present) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{1}	ptab?(pt1′)
{2}	pde_pt?(pdir(pt2′))
{3}	ptab?(pt2′)
{4}	not_present?(pdir(pt2′))

Applying decompose-equality,

This completes the proof of `set_reference_present.9`.

`set_reference_present.10`:

{-1}	ptab?(pt1′)
{-2}	not_present?(ptab(pt1′))
{-3}	Ptab(Not_present) = Ptab(Pte(pte(ptab(pt2′)) WITH [accessed := TRUE]))
{-4}	ptab?(pt2′)
{1}	Write?(access′)
{2}	not_present?(ptab(pt2′))

Applying decompose-equality,

This completes the proof of `set_reference_present.10`.

`set_reference_present.11`:

{-1}	not_present?(pdir(pt1′))
{-2}	pde_pt?(pdir(pt2′))
{-3}	Pdir(Not_present) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)

Applying decompose-equality,

This completes the proof of `set_reference_present.11`.

`set_reference_present.12`:

{-1}	not_present?(pdir(pt1′))
{-2}	Pdir(Not_present) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	pde_pt?(pdir(pt2′))
{4}	ptab?(pt2′)
{5}	not_present?(pdir(pt2′))

Applying decompose-equality,

This completes the proof of `set_reference_present.12`.

Q.E.D.

C.159.9 Paging_type_helpers.set_reference_base_TCC1

Terse proof for `set_reference_base_TCC1`.

`set_reference_base_TCC1:`

{1} ∀ (pt1, pt2: Paging_type, access: Memory_access):
set_reference(pt1, access) = set_reference(pt2, access) ∧ present?(pt1) ⊃ present?(pt2)

Repeatedly Skolemizing and flattening,
Using lemma set_reference_present,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `set_reference_base_TCC1`.
Q.E.D.

C.159.10 Paging_type_helpers.set_reference_base

Terse proof for `set_reference_base`.

`set_reference_base:`

{1} ∀ (pt1, pt2: Paging_type, access: Memory_access):
set_reference(pt1, access) = set_reference(pt2, access) ∧ present?(pt1) ⊃
base(pt1) = base(pt2)

Trying repeated skolemization, instantiation, and if-lifting,
we get 16 subgoals:

`set_reference_base.1:`

{-1} Write?(access′)
{-2} ptab?(pt1′)
{-3} ptab?(pt2′)
{-4} not_present?(ptab(pt2′))
{-5} Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)

{1} not_present?(ptab(pt1′))
{2} page_base(pte(ptab(pt1′))) = page_base(pte(ptab(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.1`.

`set_reference_base.2:`

{-1} Write?(access′)
{-2} ptab?(pt1′)
{-3} ptab?(pt2′)
{-4} Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
Ptab(Pte(pte(ptab(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))

{1} not_present?(ptab(pt1′))
{2} not_present?(ptab(pt2′))
{3} page_base(pte(ptab(pt1′))) = page_base(pte(ptab(pt2′)))

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_base.2`.

`set_reference_base.3:`

{-1} Write?(access′)
{-2} pde_pt?(pdir(pt1′))
{-3} not_present?(pdir(pt2′))
{-4} Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)

{1} ptab?(pt1′)
{2} ptab?(pt2′)
{3} page_table_base(pde_pt(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.3`.
`set_reference_base.4:`

{-1} Write?(access′)
{-2} pde_pt?(pdir(pt1′))
{-3} pde_pt?(pdir(pt2′))
{-4} Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))

{1} ptab?(pt1′)
{2} ptab?(pt2′)
{3} page_table_base(pde_pt(pdir(pt1′))) = page_table_base(pde_pt(pdir(pt2′)))

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_base.4`.
`set_reference_base.5:`

{-1} Write?(access′)
{-2} pde_pt?(pdir(pt1′))
{-3} Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))

{1} ptab?(pt1′)
{2} ptab?(pt2′)
{3} not_present?(pdir(pt2′))
{4} pde_pt?(pdir(pt2′))
{5} page_table_base(pde_pt(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.5`.
`set_reference_base.6:`

{-1} Write?(access′)
{-2} not_present?(pdir(pt2′))
{-3} Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
Pdir(Not_present)

{1} ptab?(pt1′)
{2} not_present?(pdir(pt1′))
{3} pde_pt?(pdir(pt1′))
{4} ptab?(pt2′)
{5} page_base(pde_4m(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.6`.

`set_reference_base.7:`

```
{-1}  Write?(access')
{-2}  pde_pt?(pdir(pt2'))
{-3}  Pdir(Pde_4m(pde_4m(pdir(pt1')) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2')) WITH [accessed := TRUE]))
|-------
{1}   ptab?(pt1')
{2}   not_present?(pdir(pt1'))
{3}   pde_pt?(pdir(pt1'))
{4}   ptab?(pt2')
{5}   page_base(pde_4m(pdir(pt1'))) = page_table_base(pde_pt(pdir(pt2')))
```

Applying decompose-equality,
This completes the proof of `set_reference_base.7`.
`set_reference_base.8:`

```
{-1}  Write?(access')
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1')) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2')) WITH [dirty := TRUE, accessed := TRUE]))
|-------
{1}   ptab?(pt1')
{2}   not_present?(pdir(pt1'))
{3}   pde_pt?(pdir(pt1'))
{4}   ptab?(pt2')
{5}   not_present?(pdir(pt2'))
{6}   pde_pt?(pdir(pt2'))
{7}   page_base(pde_4m(pdir(pt1'))) = page_base(pde_4m(pdir(pt2')))
```

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_base.8`.
`set_reference_base.9:`

```
{-1}  ptab?(pt1')
{-2}  ptab?(pt2')
{-3}  not_present?(ptab(pt2'))
{-4}  Ptab(Pte(pte(ptab(pt1')) WITH [accessed := TRUE])) = Ptab(Not_present)
|-------
{1}   Write?(access')
{2}   not_present?(ptab(pt1'))
{3}   page_base(pte(ptab(pt1'))) = page_base(pte(ptab(pt2')))
```

Applying decompose-equality,
This completes the proof of `set_reference_base.9`.
`set_reference_base.10:`

```
{-1}  ptab?(pt1')
{-2}  ptab?(pt2')
{-3}  Ptab(Pte(pte(ptab(pt1')) WITH [accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2')) WITH [accessed := TRUE]))
|-------
{1}   Write?(access')
{2}   not_present?(ptab(pt1'))
{3}   not_present?(ptab(pt2'))
{4}   page_base(pte(ptab(pt1'))) = page_base(pte(ptab(pt2')))
```

Applying decompose-equality,
Applying decompose-equality,

Applying decompose-equality,
This completes the proof of `set_reference_base.10`.
`set_reference_base.11`:

{-1}	pde_pt?(pdir(pt1′))
{-2}	not_present?(pdir(pt2′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	page_table_base(pde_pt(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.11`.
`set_reference_base.12`:

{-1}	pde_pt?(pdir(pt1′))
{-2}	pde_pt?(pdir(pt2′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	page_table_base(pde_pt(pdir(pt1′))) = page_table_base(pde_pt(pdir(pt2′)))

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_base.12`.
`set_reference_base.13`:

{-1}	pde_pt?(pdir(pt1′))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	not_present?(pdir(pt2′))
{5}	pde_pt?(pdir(pt2′))
{6}	page_table_base(pde_pt(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.13`.
`set_reference_base.14`:

{-1}	not_present?(pdir(pt2′))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	page_base(pde_4m(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))

Applying decompose-equality,
This completes the proof of `set_reference_base.14`.

`set_reference_base.15:`

```
{-1}  pde_pt?(pdir(pt2′))
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
       Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
------------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   not_present?(pdir(pt1′))
{4}   pde_pt?(pdir(pt1′))
{5}   ptab?(pt2′)
{6}   page_base(pde_4m(pdir(pt1′))) = page_table_base(pde_pt(pdir(pt2′)))
```

Applying decompose-equality,
This completes the proof of `set_reference_base.15`.
`set_reference_base.16:`

```
{-1}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
       Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
------------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   not_present?(pdir(pt1′))
{4}   pde_pt?(pdir(pt1′))
{5}   ptab?(pt2′)
{6}   not_present?(pdir(pt2′))
{7}   pde_pt?(pdir(pt2′))
{8}   page_base(pde_4m(pdir(pt1′))) = page_base(pde_4m(pdir(pt2′)))
```

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_base.16`.
Q.E.D.

C.159.11 Paging_type_helpers.set_reference_accessible

Terse proof for `set_reference_accessible`.
`set_reference_accessible:`

```
------------------------------------------------------------------
{1}   ∀ (pt1, pt2: Paging_type, access, ac1: Memory_access):
        set_reference(pt1, access) = set_reference(pt2, access) ∧ present?(pt1) ⊃
         accessible?(pt1, ac1) = accessible?(pt2, ac1)
```

Trying repeated skolemization, instantiation, and if-lifting,
we get 32 subgoals:
`set_reference_accessible.1:`

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  not_present?(ptab(pt2′))
{-5}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}  access(pte(ptab(pt1′)))(ac1′)
------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   access(pte(ptab(pt2′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.1`.

`set_reference_accessible.2`:

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-5}  access(pte(ptab(pt1′)))(ac1′)
------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   not_present?(ptab(pt2′))
{3}   access(pte(ptab(pt2′)))(ac1′)
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.2`.

`set_reference_accessible.3`:

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  not_present?(ptab(pt2′))
{-5}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}  access(pte(ptab(pt2′)))(ac1′)
------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   access(pte(ptab(pt1′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.3`.

`set_reference_accessible.4`:

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-5}  access(pte(ptab(pt2′)))(ac1′)
------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   not_present?(ptab(pt2′))
{3}   access(pte(ptab(pt1′)))(ac1′)
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.4`.

`set_reference_accessible.5`:

```
  {-1}  Write?(access')
  {-2}  pde_pt?(pdir(pt1'))
  {-3}  not_present?(pdir(pt2'))
  {-4}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) = Pdir(Not_present)
  {-5}  access(pde_4m(pdir(pt2')))(ac1')
  |-------
  {1}   ptab?(pt1')
  {2}   ptab?(pt2')
  {3}   access(pde_pt(pdir(pt1')))(ac1')
```

Applying decompose-equality,
This completes the proof of `set_reference_accessible.5`.
`set_reference_accessible.6`:

```
  {-1}  Write?(access')
  {-2}  pde_pt?(pdir(pt1'))
  {-3}  pde_pt?(pdir(pt2'))
  {-4}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) =
          Pdir(Pde_pt(pde_pt(pdir(pt2')) WITH [accessed := TRUE]))
  {-5}  access(pde_pt(pdir(pt2')))(ac1')
  |-------
  {1}   ptab?(pt1')
  {2}   ptab?(pt2')
  {3}   access(pde_pt(pdir(pt1')))(ac1')
```

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_accessible.6`.
`set_reference_accessible.7`:

```
  {-1}  Write?(access')
  {-2}  pde_pt?(pdir(pt1'))
  {-3}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) =
          Pdir(Pde_4m(pde_4m(pdir(pt2')) WITH [dirty := TRUE, accessed := TRUE]))
  {-4}  access(pde_4m(pdir(pt2')))(ac1')
  |-------
  {1}   ptab?(pt1')
  {2}   ptab?(pt2')
  {3}   not_present?(pdir(pt2'))
  {4}   pde_pt?(pdir(pt2'))
  {5}   access(pde_pt(pdir(pt1')))(ac1')
```

Applying decompose-equality,
This completes the proof of `set_reference_accessible.7`.
`set_reference_accessible.8`:

```
  {-1}  Write?(access')
  {-2}  pde_pt?(pdir(pt1'))
  {-3}  not_present?(pdir(pt2'))
  {-4}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) = Pdir(Not_present)
  {-5}  access(pde_pt(pdir(pt1')))(ac1')
  |-------
  {1}   ptab?(pt1')
  {2}   ptab?(pt2')
  {3}   access(pde_4m(pdir(pt2')))(ac1')
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.8`.

`set_reference_accessible.9:`

{-1}	Write?(access′)
{-2}	pde_pt?(pdir(pt1′))
{-3}	pde_pt?(pdir(pt2′))
{-4}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-5}	access(pde_pt(pdir(pt1′)))(ac1′)
{1}	ptab?(pt1′)
{2}	ptab?(pt2′)
{3}	access(pde_pt(pdir(pt2′)))(ac1′)

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.9`.

`set_reference_accessible.10:`

{-1}	Write?(access′)
{-2}	pde_pt?(pdir(pt1′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-4}	access(pde_pt(pdir(pt1′)))(ac1′)
{1}	ptab?(pt1′)
{2}	ptab?(pt2′)
{3}	not_present?(pdir(pt2′))
{4}	pde_pt?(pdir(pt2′))
{5}	access(pde_4m(pdir(pt2′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.10`.

`set_reference_accessible.11:`

{-1}	Write?(access′)
{-2}	not_present?(pdir(pt2′))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{-4}	access(pde_4m(pdir(pt2′)))(ac1′)
{1}	ptab?(pt1′)
{2}	not_present?(pdir(pt1′))
{3}	pde_pt?(pdir(pt1′))
{4}	ptab?(pt2′)
{5}	access(pde_4m(pdir(pt1′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.11`.

`set_reference_accessible.12:`

```
{-1}  Write?(access′)
{-2}  pde_pt?(pdir(pt2′))
{-3}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}  access(pde_pt(pdir(pt2′)))(ac1′)
-------------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   not_present?(pdir(pt1′))
{3}   pde_pt?(pdir(pt1′))
{4}   ptab?(pt2′)
{5}   access(pde_4m(pdir(pt1′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.12`.

`set_reference_accessible.13:`

```
{-1}  Write?(access′)
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-3}  access(pde_4m(pdir(pt2′)))(ac1′)
-------------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   not_present?(pdir(pt1′))
{3}   pde_pt?(pdir(pt1′))
{4}   ptab?(pt2′)
{5}   not_present?(pdir(pt2′))
{6}   pde_pt?(pdir(pt2′))
{7}   access(pde_4m(pdir(pt1′)))(ac1′)
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.13`.

`set_reference_accessible.14:`

```
{-1}  Write?(access′)
{-2}  not_present?(pdir(pt2′))
{-3}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Not_present)
{-4}  access(pde_4m(pdir(pt1′)))(ac1′)
-------------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   not_present?(pdir(pt1′))
{3}   pde_pt?(pdir(pt1′))
{4}   ptab?(pt2′)
{5}   access(pde_4m(pdir(pt2′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.14`.

`set_reference_accessible.15:`

{-1}	Write?(access′)
{-2}	pde_pt?(pdir(pt2′))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}	access(pde_4m(pdir(pt1′)))(ac1′)
{1}	ptab?(pt1′)
{2}	not_present?(pdir(pt1′))
{3}	pde_pt?(pdir(pt1′))
{4}	ptab?(pt2′)
{5}	access(pde_pt(pdir(pt2′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.15`.

`set_reference_accessible.16:`

{-1}	Write?(access′)
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-3}	access(pde_4m(pdir(pt1′)))(ac1′)
{1}	ptab?(pt1′)
{2}	not_present?(pdir(pt1′))
{3}	pde_pt?(pdir(pt1′))
{4}	ptab?(pt2′)
{5}	not_present?(pdir(pt2′))
{6}	pde_pt?(pdir(pt2′))
{7}	access(pde_4m(pdir(pt2′)))(ac1′)

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.16`.

`set_reference_accessible.17:`

{-1}	ptab?(pt1′)
{-2}	ptab?(pt2′)
{-3}	not_present?(ptab(pt2′))
{-4}	Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}	access(pte(ptab(pt1′)))(ac1′)
{1}	Write?(access′)
{2}	not_present?(ptab(pt1′))
{3}	access(pte(ptab(pt2′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.17`.

`set_reference_accessible.18:`

```
{-1}  ptab?(pt1′)
{-2}  ptab?(pt2′)
{-3}  Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2′)) WITH [accessed := TRUE]))
{-4}  access(pte(ptab(pt1′)))(ac1′)
------------------------------------------------------------
{1}   Write?(access′)
{2}   not_present?(ptab(pt1′))
{3}   not_present?(ptab(pt2′))
{4}   access(pte(ptab(pt2′)))(ac1′)
```

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_accessible.18`.

`set_reference_accessible.19:`

```
{-1}  ptab?(pt1′)
{-2}  ptab?(pt2′)
{-3}  not_present?(ptab(pt2′))
{-4}  Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}  access(pte(ptab(pt2′)))(ac1′)
------------------------------------------------------------
{1}   Write?(access′)
{2}   not_present?(ptab(pt1′))
{3}   access(pte(ptab(pt1′)))(ac1′)
```

Applying decompose-equality,
This completes the proof of `set_reference_accessible.19`.

`set_reference_accessible.20:`

```
{-1}  ptab?(pt1′)
{-2}  ptab?(pt2′)
{-3}  Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2′)) WITH [accessed := TRUE]))
{-4}  access(pte(ptab(pt2′)))(ac1′)
------------------------------------------------------------
{1}   Write?(access′)
{2}   not_present?(ptab(pt1′))
{3}   not_present?(ptab(pt2′))
{4}   access(pte(ptab(pt1′)))(ac1′)
```

Applying decompose-equality,
Applying decompose-equality,
Applying decompose-equality,
This completes the proof of `set_reference_accessible.20`.

`set_reference_accessible.21:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  not_present?(pdir(pt2′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}  access(pde_4m(pdir(pt2′)))(ac1′)
------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
{4}   access(pde_pt(pdir(pt1′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.21`.

`set_reference_accessible.22:`

{-1}	pde_pt?(pdir(pt1′))
{-2}	pde_pt?(pdir(pt2′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}	access(pde_pt(pdir(pt2′)))(ac1′)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	access(pde_pt(pdir(pt1′)))(ac1′)

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.22`.

`set_reference_accessible.23:`

{-1}	pde_pt?(pdir(pt1′))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}	access(pde_4m(pdir(pt2′)))(ac1′)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	not_present?(pdir(pt2′))
{5}	pde_pt?(pdir(pt2′))
{6}	access(pde_pt(pdir(pt1′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.23`.

`set_reference_accessible.24:`

{-1}	pde_pt?(pdir(pt1′))
{-2}	not_present?(pdir(pt2′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}	access(pde_pt(pdir(pt1′)))(ac1′)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	access(pde_4m(pdir(pt2′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.24`.

`set_reference_accessible.25:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  pde_pt?(pdir(pt2′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
       Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}  access(pde_pt(pdir(pt1′)))(ac1′)
|-------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
{4}   access(pde_pt(pdir(pt2′)))(ac1′)
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.25`.

`set_reference_accessible.26:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
       Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}  access(pde_pt(pdir(pt1′)))(ac1′)
|-------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
{4}   not_present?(pdir(pt2′))
{5}   pde_pt?(pdir(pt2′))
{6}   access(pde_4m(pdir(pt2′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.26`.

`set_reference_accessible.27:`

```
{-1}  not_present?(pdir(pt2′))
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}  access(pde_4m(pdir(pt2′)))(ac1′)
|-------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   not_present?(pdir(pt1′))
{4}   pde_pt?(pdir(pt1′))
{5}   ptab?(pt2′)
{6}   access(pde_4m(pdir(pt1′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.27`.

`set_reference_accessible.28:`

{-1}	pde_pt?(pdir(pt2′))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessible := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessible := TRUE]))
{-3}	access(pde_pt(pdir(pt2′)))(ac1′)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	access(pde_4m(pdir(pt1′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.28`.

`set_reference_accessible.29:`

{-1}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessible := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessible := TRUE]))
{-2}	access(pde_4m(pdir(pt2′)))(ac1′)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	not_present?(pdir(pt2′))
{7}	pde_pt?(pdir(pt2′))
{8}	access(pde_4m(pdir(pt1′)))(ac1′)

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.29`.

`set_reference_accessible.30:`

{-1}	not_present?(pdir(pt2′))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessible := TRUE])) = Pdir(Not_present)
{-3}	access(pde_4m(pdir(pt1′)))(ac1′)
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	access(pde_4m(pdir(pt2′)))(ac1′)

Applying decompose-equality,

This completes the proof of `set_reference_accessible.30`.

`set_reference_accessible.31:`

```
{-1}  pde_pt?(pdir(pt2′))
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}  access(pde_4m(pdir(pt1′)))(ac1′)
-------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   not_present?(pdir(pt1′))
{4}   pde_pt?(pdir(pt1′))
{5}   ptab?(pt2′)
{6}   access(pde_pt(pdir(pt2′)))(ac1′)
```

Applying decompose-equality,

This completes the proof of `set_reference_accessible.31`.

`set_reference_accessible.32:`

```
{-1}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-2}  access(pde_4m(pdir(pt1′)))(ac1′)
-------------------------------------------------------------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   not_present?(pdir(pt1′))
{4}   pde_pt?(pdir(pt1′))
{5}   ptab?(pt2′)
{6}   not_present?(pdir(pt2′))
{7}   pde_pt?(pdir(pt2′))
{8}   access(pde_4m(pdir(pt2′)))(ac1′)
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_accessible.32`.

Q.E.D.

C.159.12 Paging_type_helpers.set_reference_privileged

Terse proof for `set_reference_privileged`.

`set_reference_privileged:`

```
{1}  ∀ (pt1, pt2: Paging_type, access: Memory_access, priv: Memory_privilege):
       set_reference(pt1, access) = set_reference(pt2, access) ∧ present?(pt1) ⊃
         privileged?(pt1, priv) = privileged?(pt2, priv)
```

Trying repeated skolemization, instantiation, and if-lifting,

we get 32 subgoals:

`set_reference_privileged.1:`

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  not_present?(ptab(pt2′))
{-5}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}  User?(privilege(pte(ptab(pt1′))))
------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   Supervisor?(priv′)
{3}   User?(privilege(pte(ptab(pt2′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.1`.

`set_reference_privileged.2:`

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-5}  User?(privilege(pte(ptab(pt1′))))
------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   not_present?(ptab(pt2′))
{3}   Supervisor?(priv′)
{4}   User?(privilege(pte(ptab(pt2′))))
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.2`.

`set_reference_privileged.3:`

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  not_present?(ptab(pt2′))
{-5}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Ptab(Not_present)
{-6}  User?(privilege(pte(ptab(pt2′))))
------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   User?(privilege(pte(ptab(pt1′))))
{3}   Supervisor?(priv′)
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.3`.

`set_reference_privileged.4`:

```
{-1}  Write?(access′)
{-2}  ptab?(pt1′)
{-3}  ptab?(pt2′)
{-4}  Ptab(Pte(pte(ptab(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-5}  User?(privilege(pte(ptab(pt2′))))
-------------------------------------------------------------------------
{1}   not_present?(ptab(pt1′))
{2}   not_present?(ptab(pt2′))
{3}   User?(privilege(pte(ptab(pt1′))))
{4}   Supervisor?(priv′)
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.4`.

`set_reference_privileged.5`:

```
{-1}  Write?(access′)
{-2}  pde_pt?(pdir(pt1′))
{-3}  not_present?(pdir(pt2′))
{-4}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-5}  User?(privilege(pde_4m(pdir(pt2′))))
-------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   ptab?(pt2′)
{3}   Supervisor?(priv′)
{4}   User?(privilege(pde_pt(pdir(pt1′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.5`.

`set_reference_privileged.6`:

```
{-1}  Write?(access′)
{-2}  pde_pt?(pdir(pt1′))
{-3}  pde_pt?(pdir(pt2′))
{-4}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-5}  User?(privilege(pde_pt(pdir(pt2′))))
-------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   ptab?(pt2′)
{3}   Supervisor?(priv′)
{4}   User?(privilege(pde_pt(pdir(pt1′))))
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.6`.

`set_reference_privileged.7:`

```
{-1}  Write?(access')
{-2}  pde_pt?(pdir(pt1'))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2')) WITH [dirty := TRUE, accessed := TRUE]))
{-4}  User?(privilege(pde_4m(pdir(pt2'))))
------------------------------------------------------------------------
{1}   ptab?(pt1')
{2}   ptab?(pt2')
{3}   not_present?(pdir(pt2'))
{4}   pde_pt?(pdir(pt2'))
{5}   Supervisor?(priv')
{6}   User?(privilege(pde_pt(pdir(pt1'))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.7`.

`set_reference_privileged.8:`

```
{-1}  Write?(access')
{-2}  pde_pt?(pdir(pt1'))
{-3}  not_present?(pdir(pt2'))
{-4}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) = Pdir(Not_present)
{-5}  User?(privilege(pde_pt(pdir(pt1'))))
------------------------------------------------------------------------
{1}   ptab?(pt1')
{2}   ptab?(pt2')
{3}   Supervisor?(priv')
{4}   User?(privilege(pde_4m(pdir(pt2'))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.8`.

`set_reference_privileged.9:`

```
{-1}  Write?(access')
{-2}  pde_pt?(pdir(pt1'))
{-3}  pde_pt?(pdir(pt2'))
{-4}  Pdir(Pde_pt(pde_pt(pdir(pt1')) WITH [accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2')) WITH [accessed := TRUE]))
{-5}  User?(privilege(pde_pt(pdir(pt1'))))
------------------------------------------------------------------------
{1}   ptab?(pt1')
{2}   ptab?(pt2')
{3}   Supervisor?(priv')
{4}   User?(privilege(pde_pt(pdir(pt2'))))
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.9`.

`set_reference_privileged.10:`

```
{-1}  Write?(access′)
{-2}  pde_pt?(pdir(pt1′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-4}  User?(privilege(pde_pt(pdir(pt1′))))
|-------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   ptab?(pt2′)
{3}   not_present?(pdir(pt2′))
{4}   pde_pt?(pdir(pt2′))
{5}   Supervisor?(priv′)
{6}   User?(privilege(pde_4m(pdir(pt2′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.10`.

`set_reference_privileged.11:`

```
{-1}  Write?(access′)
{-2}  not_present?(pdir(pt2′))
{-3}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Not_present)
{-4}  User?(privilege(pde_4m(pdir(pt2′))))
|-------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   not_present?(pdir(pt1′))
{3}   pde_pt?(pdir(pt1′))
{4}   ptab?(pt2′)
{5}   Supervisor?(priv′)
{6}   User?(privilege(pde_4m(pdir(pt1′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.11`.

`set_reference_privileged.12:`

```
{-1}  Write?(access′)
{-2}  pde_pt?(pdir(pt2′))
{-3}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}  User?(privilege(pde_pt(pdir(pt2′))))
|-------------------------------------------------------------------------
{1}   ptab?(pt1′)
{2}   not_present?(pdir(pt1′))
{3}   pde_pt?(pdir(pt1′))
{4}   ptab?(pt2′)
{5}   Supervisor?(priv′)
{6}   User?(privilege(pde_4m(pdir(pt1′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.12`.

`set_reference_privileged.13:`

{-1}	Write?(access′)
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
{-3}	User?(privilege(pde_4m(pdir(pt2′))))
{1}	ptab?(pt1′)
{2}	not_present?(pdir(pt1′))
{3}	pde_pt?(pdir(pt1′))
{4}	ptab?(pt2′)
{5}	not_present?(pdir(pt2′))
{6}	pde_pt?(pdir(pt2′))
{7}	Supervisor?(priv′)
{8}	User?(privilege(pde_4m(pdir(pt1′))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.13`.

`set_reference_privileged.14:`

{-1}	Write?(access′)
{-2}	not_present?(pdir(pt2′))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Not_present)
{-4}	User?(privilege(pde_4m(pdir(pt1′))))
{1}	ptab?(pt1′)
{2}	not_present?(pdir(pt1′))
{3}	pde_pt?(pdir(pt1′))
{4}	ptab?(pt2′)
{5}	Supervisor?(priv′)
{6}	User?(privilege(pde_4m(pdir(pt2′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.14`.

`set_reference_privileged.15:`

{-1}	Write?(access′)
{-2}	pde_pt?(pdir(pt2′))
{-3}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}	User?(privilege(pde_4m(pdir(pt1′))))
{1}	ptab?(pt1′)
{2}	not_present?(pdir(pt1′))
{3}	pde_pt?(pdir(pt1′))
{4}	ptab?(pt2′)
{5}	Supervisor?(priv′)
{6}	User?(privilege(pde_pt(pdir(pt2′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.15`.

`set_reference_privileged.16:`

```
{-1}  Write?(access')
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1'))) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2'))) WITH [dirty := TRUE, accessed := TRUE]))
{-3}  User?(privilege(pde_4m(pdir(pt1'))))
-------------------------------------------------------------------------------
{1}   ptab?(pt1')
{2}   not_present?(pdir(pt1'))
{3}   pde_pt?(pdir(pt1'))
{4}   ptab?(pt2')
{5}   not_present?(pdir(pt2'))
{6}   pde_pt?(pdir(pt2'))
{7}   Supervisor?(priv')
{8}   User?(privilege(pde_4m(pdir(pt2'))))
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.16`.

`set_reference_privileged.17:`

```
{-1}  ptab?(pt1')
{-2}  ptab?(pt2')
{-3}  not_present?(ptab(pt2'))
{-4}  Ptab(Pte(pte(ptab(pt1')) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}  User?(privilege(pte(ptab(pt1'))))
-------------------------------------------------------------------------------
{1}   Write?(access')
{2}   not_present?(ptab(pt1'))
{3}   Supervisor?(priv')
{4}   User?(privilege(pte(ptab(pt2'))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.17`.

`set_reference_privileged.18:`

```
{-1}  ptab?(pt1')
{-2}  ptab?(pt2')
{-3}  Ptab(Pte(pte(ptab(pt1')) WITH [accessed := TRUE])) =
        Ptab(Pte(pte(ptab(pt2')) WITH [accessed := TRUE]))
{-4}  User?(privilege(pte(ptab(pt1'))))
-------------------------------------------------------------------------------
{1}   Write?(access')
{2}   not_present?(ptab(pt1'))
{3}   not_present?(ptab(pt2'))
{4}   Supervisor?(priv')
{5}   User?(privilege(pte(ptab(pt2'))))
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.18`.

`set_reference_privileged.19:`

{-1}	ptab?(pt1′)
{-2}	ptab?(pt2′)
{-3}	not_present?(ptab(pt2′))
{-4}	Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) = Ptab(Not_present)
{-5}	User?(privilege(pte(ptab(pt2′))))
{1}	Write?(access′)
{2}	not_present?(ptab(pt1′))
{3}	User?(privilege(pte(ptab(pt1′))))
{4}	Supervisor?(priv′)

Applying decompose-equality,

This completes the proof of `set_reference_privileged.19`.

`set_reference_privileged.20:`

{-1}	ptab?(pt1′)
{-2}	ptab?(pt2′)
{-3}	Ptab(Pte(pte(ptab(pt1′)) WITH [accessed := TRUE])) = Ptab(Pte(pte(ptab(pt2′)) WITH [accessed := TRUE]))
{-4}	User?(privilege(pte(ptab(pt2′))))
{1}	Write?(access′)
{2}	not_present?(ptab(pt1′))
{3}	not_present?(ptab(pt2′))
{4}	User?(privilege(pte(ptab(pt1′))))
{5}	Supervisor?(priv′)

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.20`.

`set_reference_privileged.21:`

{-1}	pde_pt?(pdir(pt1′))
{-2}	not_present?(pdir(pt2′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}	User?(privilege(pde_4m(pdir(pt2′))))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	Supervisor?(priv′)
{5}	User?(privilege(pde_pt(pdir(pt1′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.21`.

`set_reference_privileged.22:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  pde_pt?(pdir(pt2′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}  User?(privilege(pde_pt(pdir(pt2′))))
|-------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
{4}   Supervisor?(priv′)
{5}   User?(privilege(pde_pt(pdir(pt1′))))
```

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.22`.

`set_reference_privileged.23:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}  User?(privilege(pde_4m(pdir(pt2′))))
|-------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
{4}   not_present?(pdir(pt2′))
{5}   pde_pt?(pdir(pt2′))
{6}   Supervisor?(priv′)
{7}   User?(privilege(pde_pt(pdir(pt1′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.23`.

`set_reference_privileged.24:`

```
{-1}  pde_pt?(pdir(pt1′))
{-2}  not_present?(pdir(pt2′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-4}  User?(privilege(pde_pt(pdir(pt1′))))
|-------
{1}   Write?(access′)
{2}   ptab?(pt1′)
{3}   ptab?(pt2′)
{4}   Supervisor?(priv′)
{5}   User?(privilege(pde_4m(pdir(pt2′))))
```

Applying decompose-equality,

This completes the proof of `set_reference_privileged.24`.

`set_reference_privileged.25:`

{-1}	pde_pt?(pdir(pt1′))
{-2}	pde_pt?(pdir(pt2′))
{-3}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}	User?(privilege(pde_pt(pdir(pt1′))))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	Supervisor?(priv′)
{5}	User?(privilege(pde_pt(pdir(pt2′))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.25`.

`set_reference_privileged.26:`

{-1}	pde_pt?(pdir(pt1′))
{-2}	Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}	User?(privilege(pde_pt(pdir(pt1′))))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	ptab?(pt2′)
{4}	not_present?(pdir(pt2′))
{5}	pde_pt?(pdir(pt2′))
{6}	Supervisor?(priv′)
{7}	User?(privilege(pde_4m(pdir(pt2′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.26`.

`set_reference_privileged.27:`

{-1}	not_present?(pdir(pt2′))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3}	User?(privilege(pde_4m(pdir(pt2′))))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	Supervisor?(priv′)
{7}	User?(privilege(pde_4m(pdir(pt1′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.27`.

`set_reference_privileged.28:`

{-1} pde_pt?(pdir(pt2′))
{-2} Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-3} User?(privilege(pde_pt(pdir(pt2′))))

{1} Write?(access′)
{2} ptab?(pt1′)
{3} not_present?(pdir(pt1′))
{4} pde_pt?(pdir(pt1′))
{5} ptab?(pt2′)
{6} Supervisor?(priv′)
{7} User?(privilege(pde_4m(pdir(pt1′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.28`.

`set_reference_privileged.29:`

{-1} Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-2} User?(privilege(pde_4m(pdir(pt2′))))

{1} Write?(access′)
{2} ptab?(pt1′)
{3} not_present?(pdir(pt1′))
{4} pde_pt?(pdir(pt1′))
{5} ptab?(pt2′)
{6} not_present?(pdir(pt2′))
{7} pde_pt?(pdir(pt2′))
{8} Supervisor?(priv′)
{9} User?(privilege(pde_4m(pdir(pt1′))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.29`.

`set_reference_privileged.30:`

{-1} not_present?(pdir(pt2′))
{-2} Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
{-3} User?(privilege(pde_4m(pdir(pt1′))))

{1} Write?(access′)
{2} ptab?(pt1′)
{3} not_present?(pdir(pt1′))
{4} pde_pt?(pdir(pt1′))
{5} ptab?(pt2′)
{6} Supervisor?(priv′)
{7} User?(privilege(pde_4m(pdir(pt2′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.30`.

`set_reference_privileged.31:`

{-1}	pde_pt?(pdir(pt2′))
{-2}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}	User?(privilege(pde_4m(pdir(pt1′))))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	Supervisor?(priv′)
{7}	User?(privilege(pde_pt(pdir(pt2′))))

Applying decompose-equality,

This completes the proof of `set_reference_privileged.31`.

`set_reference_privileged.32:`

{-1}	Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
{-2}	User?(privilege(pde_4m(pdir(pt1′))))
{1}	Write?(access′)
{2}	ptab?(pt1′)
{3}	not_present?(pdir(pt1′))
{4}	pde_pt?(pdir(pt1′))
{5}	ptab?(pt2′)
{6}	not_present?(pdir(pt2′))
{7}	pde_pt?(pdir(pt2′))
{8}	Supervisor?(priv′)
{9}	User?(privilege(pde_4m(pdir(pt2′))))

Applying decompose-equality,

Applying decompose-equality,

Applying decompose-equality,

This completes the proof of `set_reference_privileged.32`.

Q.E.D.

C.159.13 Paging_type_helpers.set_reference_leaf

Terse proof for `set_reference_leaf`.

`set_reference_leaf:`

{1}	∀ (pt1, pt2: Paging_type, access: Memory_access): set_reference(pt1, access) = set_reference(pt2, access) ∧ present?(pt1) ⊃ is_leaf?(pt1) = is_leaf?(pt2)

Trying repeated skolemization, instantiation, and if-lifting,

we get 6 subgoals:

`set_reference_leaf.1:`

```
{-1}  Write?(access′)
{-2}  pdir?(pt1′)
{-3}  pde_pt?(pdir(pt1′))
{-4}  not_present?(pdir(pt2′))
{-5}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
|-----
{1}   ptab?(pt2′)
```

Applying decompose-equality,

This completes the proof of `set_reference_leaf.1`.

`set_reference_leaf.2:`

```
{-1}  Write?(access′)
{-2}  pdir?(pt1′)
{-3}  pde_pt?(pdir(pt1′))
{-4}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [dirty := TRUE, accessed := TRUE]))
|-----
{1}   ptab?(pt2′)
{2}   not_present?(pdir(pt2′))
{3}   pde_pt?(pdir(pt2′))
```

Applying decompose-equality,

This completes the proof of `set_reference_leaf.2`.

`set_reference_leaf.3:`

```
{-1}  Write?(access′)
{-2}  pdir?(pt1′)
{-3}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [dirty := TRUE, accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-4}  pdir?(pt2′)
{-5}  pde_pt?(pdir(pt2′))
|-----
{1}   not_present?(pdir(pt1′))
{2}   pde_pt?(pdir(pt1′))
```

Applying decompose-equality,

This completes the proof of `set_reference_leaf.3`.

`set_reference_leaf.4:`

```
{-1}  pdir?(pt1′)
{-2}  pde_pt?(pdir(pt1′))
{-3}  not_present?(pdir(pt2′))
{-4}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) = Pdir(Not_present)
|-----
{1}   Write?(access′)
{2}   ptab?(pt2′)
```

Applying decompose-equality,

This completes the proof of `set_reference_leaf.4`.

`set_reference_leaf.5:`

```
{-1}  pdir?(pt1′)
{-2}  pde_pt?(pdir(pt1′))
{-3}  Pdir(Pde_pt(pde_pt(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_4m(pde_4m(pdir(pt2′)) WITH [accessed := TRUE]))
|-------
{1}   Write?(access′)
{2}   ptab?(pt2′)
{3}   not_present?(pdir(pt2′))
{4}   pde_pt?(pdir(pt2′))
```

Applying decompose-equality,
This completes the proof of `set_reference_leaf.5`.
`set_reference_leaf.6:`

```
{-1}  pdir?(pt1′)
{-2}  Pdir(Pde_4m(pde_4m(pdir(pt1′)) WITH [accessed := TRUE])) =
        Pdir(Pde_pt(pde_pt(pdir(pt2′)) WITH [accessed := TRUE]))
{-3}  pdir?(pt2′)
{-4}  pde_pt?(pdir(pt2′))
|-------
{1}   Write?(access′)
{2}   not_present?(pdir(pt1′))
{3}   pde_pt?(pdir(pt1′))
```

Applying decompose-equality,
This completes the proof of `set_reference_leaf.6`.
Q.E.D.

C.160 Proofs for Pde_type (paging-data.pvs)

This theory contains no provable formal statements.

C.161 Proofs for Pde_type_adt (Pde_type_adt.pvs)

C.161.1 Pde_type_adt.Pde_4m_TCC1

Terse proof for `Pde_4m_TCC1`.
`Pde_4m_TCC1:`

```
|-------
{1}   ∀ (a: Memory_Address_4G): offset(a) ≥ 0
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Pde_4m_TCC1`.
Q.E.D.

C.162 Proofs for Pde_type_adt_reduce (Pde_type_adt.pvs)

This theory contains no provable formal statements.

C.163 Proofs for Phy_Mem_Blessing (challenge-phymem.pvs)

C.163.1 Phy_Mem_Blessing.phy_mem_plain_memory

Terse proof for `phy_mem_plain_memory`.

`phy_mem_plain_memory:`

```
{1}  ∀ (oact: PRED[[Physical_memory → SuperResult[Physical_memory]]]):
       unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact, in_memory(min, max))
         ⊃ plain_memory?(phy_pm(oact))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of plain_memory?,
Expanding the definition of phy_pm,
Rewriting using union_empty_left, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 7 subgoals:

`phy_mem_plain_memory.1:`

```
{-1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, max))
{1}   side_effect_content_unchanged(in_memory(min, max), fullset[Physical_memory],
                                    memory_write_side_effect(phy_mem))
```

Hiding formulas: (-1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `phy_mem_plain_memory.1`.

`phy_mem_plain_memory.2:`

```
{-1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, max))
{1}   side_effect_content_unchanged(in_memory(min, max), fullset[Physical_memory],
                                    memory_read_side_effect(phy_mem))
```

Hiding formulas: (-1),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `phy_mem_plain_memory.2`.

`phy_mem_plain_memory.3:`

```
{-1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, max))
{1}   transformers_ok?(fullset[Physical_memory],
                       ((memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_write_transfo
```

Hiding formulas: -1,
Using lemma phy_mem_read_transformers_ok,
Using lemma phy_mem_write_transformers_ok,
Using lemma phy_mem_read_side_effects_transformers_ok,
Using lemma phy_mem_write_side_effects_transformers_ok,
Rewriting using transformers_ok_union_transformers, matching in *,
we get 2 subgoals:

phy_mem_plain_memory.3.1:

{-1} transformers_ok?(fullset[Physical_memory[min, max]],
memory_write_side_effect_super_transformers(phy_mem, in_memory(min, m
{-2} transformers_ok?(fullset[Physical_memory[min, max]],
memory_read_side_effect_super_transformers(phy_mem, in_memory(min, ma
{-3} transformers_ok?(fullset[Physical_memory[min, max]],
memory_write_transformers(phy_mem, in_memory(min, max)))
{-4} transformers_ok?(fullset[Physical_memory[min, max]],
memory_read_transformers(phy_mem, in_memory(min, max)))

{1} transformers_ok?(fullset[Physical_memory],
(memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_wr
{2} transformers_ok?(fullset[Physical_memory],
((memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_w

Rewriting using transformers_ok_union_transformers, matching in *,
This completes the proof of phy_mem_plain_memory.3.1.
phy_mem_plain_memory.3.2:

{-1} transformers_ok?(fullset[Physical_memory[min, max]],
memory_write_side_effect_super_transformers(phy_mem, in_memory(min, m
{-2} transformers_ok?(fullset[Physical_memory[min, max]],
memory_read_side_effect_super_transformers(phy_mem, in_memory(min, ma
{-3} transformers_ok?(fullset[Physical_memory[min, max]],
memory_write_transformers(phy_mem, in_memory(min, max)))
{-4} transformers_ok?(fullset[Physical_memory[min, max]],
memory_read_transformers(phy_mem, in_memory(min, max)))

{1} transformers_ok?(fullset[Physical_memory],
(memory_read_side_effect_super_transformers(phy_mem, in_memory(min, ma
{2} transformers_ok?(fullset[Physical_memory],
((memory_read_transformers(phy_mem, in_memory(min, max)) ∪ memory_w

Rewriting using transformers_ok_union_transformers, matching in *,
This completes the proof of phy_mem_plain_memory.3.2.
phy_mem_plain_memory.4:

{-1} unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, ma

{1} changed_memory_invariant?(phy_mem, fullset[Physical_memory], in_memory(min, max))

Rewriting using phy_mem_changed_memory_invariant, matching in *,
This completes the proof of phy_mem_plain_memory.4.
phy_mem_plain_memory.5:

{-1} unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, ma

{1} unchanged_memory_write_invariant?(phy_mem, fullset[Physical_memory], in_memory(min, max

Rewriting using phy_mem_unchanged_memory_write_invariant, matching in *,
This completes the proof of phy_mem_plain_memory.5.
phy_mem_plain_memory.6:

{-1} unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, ma

{1} unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
memory_write_transformers(phy_mem, in_memory(min, max))
emptyset[Address])

Hiding formulas: -1,
Expanding the definition of unchanged_memory_invariant?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`phy_mem_plain_memory.6.1:`

{1} ∀ (s: Physical_memory[min, max],
q: [Physical_memory[min, max] → SuperResult[Physical_memory[min, max]]],
a: Address):
fullset[Physical_memory](s) ∧
memory_write_transformers(phy_mem, in_memory(min, max))(q) ∧
emptyset[Address](a) ∧
OK?(q(s)) ∧
OK?(memory_read(phy_mem)(a)(s)) ∧
OK?(memory_read(phy_mem)(a)(state(q(s))))
⊃
data(memory_read(phy_mem)(a)(state(q(s)))) =
data(memory_read(phy_mem)(a)(s))

Repeatedly Skolemizing and flattening,

Expanding the definition of emptyset,

which is trivially true.

This completes the proof of `phy_mem_plain_memory.6.1`.

`phy_mem_plain_memory.6.2:`

{1} transformer_invariant?(fullset[Physical_memory],
memory_write_transformers(phy_mem, in_memory(min, max)))

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of `phy_mem_plain_memory.6.2`.

`phy_mem_plain_memory.7:`

{-1} unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact′, in_memory(min, max))
{1} unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
((oact′ ∪ memory_read_transformers(phy_mem, in_memory(min, max))) ∪
in_memory(min, max))

Using lemma phy_mem_unchanged_memory_invariant_read_transformers,

Using lemma phy_mem_unchanged_memory_invariant_read_side_effect_transformers,

Using lemma phy_mem_unchanged_memory_invariant_write_side_effect_transformers,

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

we get 2 subgoals:

`phy_mem_plain_memory.7.1:`

```
{-1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]],
                                  memory_write_side_effect_super_transformers(phy_mem,
                                                                              in_memory
                                  in_memory(min, max))
{-2}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]],
                                  memory_read_side_effect_super_transformers(phy_mem,
                                                                             in_memory(
                                  in_memory(min, max))
{-3}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]],
                                  memory_read_transformers(phy_mem, in_memory(min, max)),
                                  in_memory(min, max))
{-4}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, ma
-------------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                  (oact' ∪ memory_read_transformers(phy_mem, in_memory(min,
                                  in_memory(min, max))
{2}   unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                  ((oact' ∪ memory_read_transformers(phy_mem, in_memory(min
                                  in_memory(min, max))
```

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

This completes the proof of `phy_mem_plain_memory.7.1`.

`phy_mem_plain_memory.7.2:`

```
{-1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]],
                                  memory_write_side_effect_super_transformers(phy_mem,
                                                                              in_memory
                                  in_memory(min, max))
{-2}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]],
                                  memory_read_side_effect_super_transformers(phy_mem,
                                                                             in_memory(
                                  in_memory(min, max))
{-3}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory[min, max]],
                                  memory_read_transformers(phy_mem, in_memory(min, max)),
                                  in_memory(min, max))
{-4}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory], oact', in_memory(min, ma
-------------------------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                  (memory_read_side_effect_super_transformers(phy_mem, in_mem
                                  in_memory(min, max))
{2}   unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                  ((oact' ∪ memory_read_transformers(phy_mem, in_memory(min
                                  in_memory(min, max))
```

Rewriting using unchanged_memory_invariant_union_transformers, matching in *,

This completes the proof of `phy_mem_plain_memory.7.2`.

Q.E.D.

C.164 Proofs for Phy_Mem_Blessing_Data (challenge-phymem.pvs)

C.164.1 Phy_Mem_Blessing_Data.in_memory_blessed_memory

Terse proof for `in_memory_blessed_memory`.

`in_memory_blessed_memory`:

```
{1}  ∀ (addr: Address, idt: (interpreted_data_type?[Data])):
       in_memory?(idt, addr, min, max) ⊃ in_blessed_memory?(idt, addr, in_memory(min, max))
```

Installing automatic rewrites from: in_memory? in_blessed_memory? subset? member address_block in_memory + < ≤

Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `in_memory_blessed_memory`.

Q.E.D.

C.165 Proofs for Phy_Mem_Challenge_Read_Other (challenge-phymem.pvs)

C.165.1 Phy_Mem_Challenge_Read_Other.read_write_other_ok

Terse proof for `read_write_other_ok`.

`read_write_other_ok`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
       in_memory?(dt1, addr1, min, max) ∧
        in_memory?(dt2, addr2, min, max) ∧
         blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
          valid_in_mem(phy_mem, dt2)(addr2)(s)
        ⊃
        OK?((write_data(phy_mem, dt1)(addr1, data1) ## read_data(phy_mem, dt2)(addr2))(s))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty?

Using lemma phy_mem_plain_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_read_write_other_ok[Physical_memory, Data1, Data2],

Installing automatic rewrites from: fullset! union_empty_left phy_pm read_data write_data valid_in_mem

Simplifying, rewriting, and recording with decision procedures,

Keeping (-4 -5 1) and hiding *,

Using lemma in_memory_blessed_memory[Data1, min, max],

Using lemma in_memory_blessed_memory[Data2, min, max],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `read_write_other_ok`.

Q.E.D.

C.165.2 Phy_Mem_Challenge_Read_Other.read_write_other_res_TCC1

Terse proof for `read_write_other_res_TCC1`.

`read_write_other_res_TCC1`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
       valid_in_mem(phy_mem, dt2)(addr2)(s) ∧
        blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
         in_memory?(dt2, addr2, min, max) ∧ in_memory?(dt1, addr1, min, max)
        ⊃
        OK?[Physical_memory[min, max], Data2]
            ((##[Physical_memory[min, max], Unit, Data2]
                  (write_data[Physical_memory[min, max], Data1]
                        (phy_mem[min, max], dt1)(addr1, data1),
                    read_data[Physical_memory[min, max], Data2]
                        (phy_mem[min, max], dt2)(addr2)))
                (s))
```

Repeatedly Skolemizing and flattening,

Rewriting using read_write_other_ok, matching in *,

This completes the proof of `read_write_other_res_TCC1`.

Q.E.D.

C.165.3 Phy_Mem_Challenge_Read_Other.read_write_other_res_TCC2

Terse proof for `read_write_other_res_TCC2`.

`read_write_other_res_TCC2`:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
       valid_in_mem(phy_mem, dt2)(addr2)(s) ∧
        blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
         in_memory?(dt2, addr2, min, max) ∧ in_memory?(dt1, addr1, min, max)
        ⊃
        OK?[Physical_memory[min, max], Data2]
            (read_data[Physical_memory[min, max], Data2]
                  (phy_mem[min, max], dt2)(addr2)(s))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty? union_empty_left phy_pm fullset in_memory_blessed_memory

Using lemma phy_mem_plain_memory,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_read_data_ok[Physical_memory, Data2],

Installing automatic rewrites from: (read_data! write_data! valid_in_mem!)

Simplifying, rewriting, and recording with decision procedures,
Rewriting using in_memory_blessed_memory[Data2, min, max], matching in *,
This completes the proof of `read_write_other_res_TCC2`.
Q.E.D.

C.165.4 Phy_Mem_Challenge_Read_Other.read_write_other_res

Terse proof for `read_write_other_res`.
`read_write_other_res`:

{1} ∀ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
in_memory?(dt1, addr1, min, max) ∧
in_memory?(dt2, addr2, min, max) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
valid_in_mem(phy_mem, dt2)(addr2)(s)
⊃
data((write_data(phy_mem, dt1)(addr1, data1) ## read_data(phy_mem, dt2)(addr2))
(s))
= data(read_data(phy_mem, dt2)(addr2)(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty? union_empty_left fullset phy_pm
Using lemma phy_mem_plain_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_write_other_res[Physical_memory, Data1, Data2],
Installing automatic rewrites from: (read_data! write_data! valid_in_mem!)
Simplifying, rewriting, and recording with decision procedures,
Rewriting using in_memory_blessed_memory[Data2, min, max], matching in *,
Rewriting using in_memory_blessed_memory[Data1, min, max], matching in *,
This completes the proof of `read_write_other_res`.
Q.E.D.

C.165.5 Phy_Mem_Challenge_Read_Other.read_read_ok

Terse proof for `read_read_ok`.
`read_read_ok`:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
in_memory?(dt1, addr1, min, max) ∧
in_memory?(dt2, addr2, min, max) ∧
valid_in_mem(phy_mem, dt1)(addr1)(s) ∧ valid_in_mem(phy_mem, dt2)(addr2)(s)
⊃ OK?((read_data(phy_mem, dt1)(addr1) ## read_data(phy_mem, dt2)(addr2))(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty? union_empty_left fullset phy_pm
Using lemma phy_mem_plain_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_read_ok[Physical_memory, Data1, Data2],

Simplifying, rewriting, and recording with decision procedures,
Using lemma in_memory_blessed_memory[Data2, min, max],
Using lemma in_memory_blessed_memory[Data1, min, max],
Installing automatic rewrites from: (read_data! write_data! valid_in_mem!)
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `read_read_ok`.
Q.E.D.

C.165.6 Phy_Mem_Challenge_Read_Other.read_read_TCC1

Terse proof for `read_read_TCC1`.
`read_read_TCC1`:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
       valid_in_mem(phy_mem, dt2)(addr2)(s) ∧
        valid_in_mem(phy_mem, dt1)(addr1)(s) ∧
         in_memory?(dt2, addr2, min, max) ∧ in_memory?(dt1, addr1, min, max)
        ⊃
        OK?[Physical_memory[min, max], Data2]
            ((##[Physical_memory[min, max], Data1, Data2]
                  (read_data[Physical_memory[min, max], Data1]
                        (phy_mem[min, max], dt1)(addr1),
                    read_data[Physical_memory[min, max], Data2]
                        (phy_mem[min, max], dt2)(addr2)))
                (s))
```

Repeatedly Skolemizing and flattening,
Rewriting using read_read_ok, matching in *,
This completes the proof of `read_read_TCC1`.
Q.E.D.

C.165.7 Phy_Mem_Challenge_Read_Other.read_read_TCC2

Terse proof for `read_read_TCC2`.
`read_read_TCC2`:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
       valid_in_mem(phy_mem, dt2)(addr2)(s) ∧
        valid_in_mem(phy_mem, dt1)(addr1)(s) ∧
         in_memory?(dt2, addr2, min, max) ∧ in_memory?(dt1, addr1, min, max)
        ⊃
        OK?[Physical_memory[min, max], Data2]
            (read_data[Physical_memory[min, max], Data2]
                (phy_mem[min, max], dt2)(addr2)(s))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty? union_empty_left fullset phy_pm
Using lemma phy_mem_plain_memory,

Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_data_ok[Physical_memory, Data2],
Installing automatic rewrites from: (read_data! write_data! valid_in_mem!)
Simplifying, rewriting, and recording with decision procedures,
Rewriting using in_memory_blessed_memory[Data2, min, max], matching in *,
This completes the proof of `read_read_TCC2`.
Q.E.D.

C.165.8 Phy_Mem_Challenge_Read_Other.read_read

Terse proof for `read_read`.
`read_read`:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), s: Physical_memory[min, max]):
in_memory?(dt1, addr1, min, max) ∧
in_memory?(dt2, addr2, min, max) ∧
valid_in_mem(phy_mem, dt1)(addr1)(s) ∧ valid_in_mem(phy_mem, dt2)(addr2)(s)
⊃
data((read_data(phy_mem, dt1)(addr1) ## read_data(phy_mem, dt2)(addr2))(s)) =
data(read_data(phy_mem, dt2)(addr2)(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty? union_empty_left fullset phy_pm
Using lemma phy_mem_plain_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_read[Physical_memory, Data1, Data2],
Installing automatic rewrites from: (read_data! write_data! valid_in_mem!)
Simplifying, rewriting, and recording with decision procedures,
Rewriting using in_memory_blessed_memory[Data1, min, max], matching in *,
Rewriting using in_memory_blessed_memory[Data2, min, max], matching in *,
This completes the proof of `read_read`.
Q.E.D.

C.166 Proofs for Phy_Mem_Challenge_Read_Same (challenge-phymem.pvs)

C.166.1 Phy_Mem_Challenge_Read_Same.read_write_ok

Terse proof for `read_write_ok`.
`read_write_ok`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]),
s: Physical_memory[min, max]):
in_memory?(dt, addr, min, max) ⊃
OK?((write_data(phy_mem, dt)(addr, data) ## read_data(phy_mem, dt)(addr))(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty?

Using lemma phy_mem_plain_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_write_ok[Physical_memory, Data],
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -2,
Expanding the definition of phy_pm,
Using lemma in_memory_blessed_memory,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of fullset,
Expanding the definition of write_data,
Expanding the definition of read_data,
which is trivially true.
This completes the proof of `read_write_ok`.
Q.E.D.

C.166.2 Phy_Mem_Challenge_Read_Same.read_write_res_TCC1

Terse proof for `read_write_res_TCC1`.

`read_write_res_TCC1:`

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]),
          s: Physical_memory[min, max]):
       in_memory?(dt, addr, min, max) ⊃
        OK?[Physical_memory[min, max], Data]
            ((##[Physical_memory[min, max], Unit, Data]
                 (write_data[Physical_memory[min, max], Data]
                      (phy_mem[min, max], dt)(addr, data),
                  read_data[Physical_memory[min, max], Data]
                      (phy_mem[min, max], dt)(addr)))
                (s))
```

Repeatedly Skolemizing and flattening,
Rewriting using read_write_ok, matching in *,
This completes the proof of `read_write_res_TCC1`.
Q.E.D.

C.166.3 Phy_Mem_Challenge_Read_Same.read_write_res

Terse proof for `read_write_res`.

`read_write_res:`

```
{1}  ∀ (addr: Address, data1: Data, dt: (interpreted_data_type?[Data]),
          s: Physical_memory[min, max]):
       in_memory?(dt, addr, min, max) ⊃
        data((write_data(phy_mem, dt)(addr, data1) ## read_data(phy_mem, dt)(addr))(s))
          = data1
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: unchanged_memory_invariant_empty emptyset_is_empty?
Using lemma phy_mem_plain_memory,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_write_res[Physical_memory, Data],

Installing automatic rewrites from: phy_pm fullset in_memory_blessed_memory read_data write_data
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `read_write_res`.
Q.E.D.

C.167 Proofs for Phy_Mem_Properties (challenge-phymem.pvs)

C.167.1 Phy_Mem_Properties.min_TCC1

Terse proof for `min_TCC1`.
`min_TCC1`:

```
{1}  ∃ (x: Memory_Address): TRUE
```

Instantiating the top quantifier in 1 with the terms: Mem(0),
Expanding the definition of Mem,
which is trivially true.
This completes the proof of `min_TCC1`.
Q.E.D.

C.167.2 Phy_Mem_Properties.phy_mem_write_transformers_ok

Terse proof for `phy_mem_write_transformers_ok`.
`phy_mem_write_transformers_ok`:

```
{1}  transformers_ok?(fullset[Physical_memory],
                      memory_write_transformers(phy_mem, in_memory(min, max)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `phy_mem_write_transformers_ok`.
Q.E.D.

C.167.3 Phy_Mem_Properties.phy_mem_read_transformers_ok

Terse proof for `phy_mem_read_transformers_ok`.
`phy_mem_read_transformers_ok`:

```
{1}  transformers_ok?(fullset[Physical_memory],
                      memory_read_transformers(phy_mem, in_memory(min, max)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `phy_mem_read_transformers_ok`.
Q.E.D.

C.167.4 Phy_Mem_Properties.phy_mem_read_side_effects_transformers_ok

Terse proof for `phy_mem_read_side_effects_transformers_ok`.

phy_mem_read_side_effects_transformers_ok:

```
{1}  transformers_ok?(fullset[Physical_memory],
                      memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_read_side_effects_transformers_ok.

Q.E.D.

C.167.5 Phy_Mem_Properties.phy_mem_write_side_effects_transformers_ok

Terse proof for phy_mem_write_side_effects_transformers_ok.

phy_mem_write_side_effects_transformers_ok:

```
{1}  transformers_ok?(fullset[Physical_memory],
                      memory_write_side_effect_super_transformers(phy_mem, in_memory(min, ma
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_write_side_effects_transformers_ok.

Q.E.D.

C.167.6 Phy_Mem_Properties.phy_mem_unchanged_memory_invariant_read_transform

Terse proof for phy_mem_unchanged_memory_invariant_read_transformers.

phy_mem_unchanged_memory_invariant_read_transformers:

```
{1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                 memory_read_transformers(phy_mem, in_memory(min, max)),
                                 in_memory(min, max))
```

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_invariant_read_transformers.1:

```
{1}  transformer_invariant?(fullset[Physical_memory],
                            memory_read_transformers(phy_mem, in_memory(min, max)))
```

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_invariant_read_transformers.1.

phy_mem_unchanged_memory_invariant_read_transformers.2:

```
{1}  ∀ (s: Physical_memory[min, max],
          q: [Physical_memory[min, max] → SuperResult[Physical_memory[min, max]]],
          a: Address):
       fullset[Physical_memory](s) ∧
        memory_read_transformers(phy_mem, in_memory(min, max))(q) ∧
         in_memory(min, max)(a) ∧
          OK?(q(s)) ∧
           OK?(memory_read(phy_mem)(a)(s)) ∧
            OK?(memory_read(phy_mem)(a)(state(q(s))))
        ⊃
        data(memory_read(phy_mem)(a)(state(q(s)))) =
         data(memory_read(phy_mem)(a)(s))
```

Expanding the definition of memory_read_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -3,

Keeping (-2 -4 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_unchanged_memory_invariant_read_transformers.2.

Q.E.D.

C.167.7 Phy_Mem_Properties.phy_mem_unchanged_memory_invariant_read_side_effect_trans

Terse proof for phy_mem_unchanged_memory_invariant_read_side_effect_transformers.

phy_mem_unchanged_memory_invariant_read_side_effect_transformers:

```
{1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                 memory_read_side_effect_super_transformers(phy_mem,
                                                                            in_memory(min, max)),
                                 in_memory(min, max))
```

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_invariant_read_side_effect_transformers.1:

```
{1}  transformer_invariant?(fullset[Physical_memory],
                            memory_read_side_effect_super_transformers(phy_mem,
                                                                       in_memory(min, max)))
```

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_invariant_read_side_effect_transformers.1.

phy_mem_unchanged_memory_invariant_read_side_effect_transformers.2:

```
{1}  ∀ (s: Physical_memory[min, max],
          q: [Physical_memory[min, max] → SuperResult[Physical_memory[min, max]]],
          a: Address):
        fullset[Physical_memory](s) ∧
         memory_read_side_effect_super_transformers(phy_mem, in_memory(min, max))(q) ∧
          in_memory(min, max)(a) ∧
           OK?(q(s)) ∧
            OK?(memory_read(phy_mem)(a)(s)) ∧
             OK?(memory_read(phy_mem)(a)(state(q(s))))
         ⊃
         data(memory_read(phy_mem)(a)(state(q(s)))) =
          data(memory_read(phy_mem)(a)(s))
```

Expanding the definition of memory_read_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 -3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of phy_mem_unchanged_memory_invariant_read_side_effect_transformers.2.

Q.E.D.

C.167.8 Phy_Mem_Properties.phy_mem_unchanged_memory_invariant_write_side_effe

Terse proof for phy_mem_unchanged_memory_invariant_write_side_effect_transformers.

phy_mem_unchanged_memory_invariant_write_side_effect_transformers:

```
{1}  unchanged_memory_invariant?(phy_mem, fullset[Physical_memory],
                                 memory_write_side_effect_super_transformers(phy_mem,
                                                                             in_memory
                                 in_memory(min, max))
```

Expanding the definition of unchanged_memory_invariant?,

Applying propositional simplification,

we get 2 subgoals:

phy_mem_unchanged_memory_invariant_write_side_effect_transformers.1:

```
{1}  transformer_invariant?(fullset[Physical_memory],
                            memory_write_side_effect_super_transformers(phy_mem,
                                                                        in_memory(min,
```

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of phy_mem_unchanged_memory_invariant_write_side_effect_transformers.1.

`phy_mem_unchanged_memory_invariant_write_side_effect_transformers.2`:

```
{1}  ∀ (s: Physical_memory[min, max],
          q: [Physical_memory[min, max] → SuperResult[Physical_memory[min, max]]],
          a: Address):
       fullset[Physical_memory](s) ∧
        memory_write_side_effect_super_transformers(phy_mem, in_memory(min, max))(q) ∧
         in_memory(min, max)(a) ∧
          OK?(q(s)) ∧
           OK?(memory_read(phy_mem)(a)(s)) ∧
            OK?(memory_read(phy_mem)(a)(state(q(s))))
        ⊃
        data(memory_read(phy_mem)(a)(state(q(s)))) =
         data(memory_read(phy_mem)(a)(s))
```

Expanding the definition of memory_write_side_effect_super_transformers,

Repeatedly Skolemizing and flattening,

Replacing using formula -4,

Keeping (-2 -3 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `phy_mem_unchanged_memory_invariant_write_side_effect_transformers.2`.

Q.E.D.

C.167.9 Phy_Mem_Properties.phy_mem_unchanged_memory_write_invariant

Terse proof for `phy_mem_unchanged_memory_write_invariant`.

`phy_mem_unchanged_memory_write_invariant`:

```
{1}  unchanged_memory_write_invariant?(phy_mem, fullset[Physical_memory], in_memory(min, max))
```

Expanding the definition of unchanged_memory_write_invariant?,

Expanding the definition of unchanged_memory_invariant?,

Expanding the definition of singleton,

Repeatedly Skolemizing and flattening,

Applying propositional simplification,

we get 2 subgoals:

`phy_mem_unchanged_memory_write_invariant.1`:

```
{-1}  b′ < max_byte
{-2}  in_memory(min, max)(waddr′)
{1}   transformer_invariant?(fullset[Physical_memory],
                             {y | y = expr_2_super(memory_write(phy_mem)(waddr′, b′))})
```

Rewriting using transformer_invariant_truth, matching in *,

This completes the proof of `phy_mem_unchanged_memory_write_invariant.1`.

`phy_mem_unchanged_memory_write_invariant.2:`

{-1} $b' < \text{max\_byte}$
{-2} $\text{in\_memory}(\text{min}, \text{max})(\text{waddr}')$

{1} $\forall\ (s\colon \text{Physical\_memory}[\text{min}, \text{max}],$
$q\colon [\text{Physical\_memory}[\text{min}, \text{max}] \to \text{SuperResult}[\text{Physical\_memory}[\text{min}, \text{max}]]],$
$a\colon \text{Address}):$
$\text{fullset}[\text{Physical\_memory}](s) \wedge$
$q = \text{expr\_2\_super}(\text{memory\_write}(\text{phy\_mem})(\text{waddr}', b')) \wedge$
$\text{remove}(\text{waddr}', \text{in\_memory}(\text{min}, \text{max}))(a) \wedge$
$\text{OK?}(q(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{phy\_mem})(a)(s)) \wedge$
$\text{OK?}(\text{memory\_read}(\text{phy\_mem})(a)(\text{state}(q(s))))$
$\supset$
$\text{data}(\text{memory\_read}(\text{phy\_mem})(a)(\text{state}(q(s)))) =$
$\text{data}(\text{memory\_read}(\text{phy\_mem})(a)(s))$

Repeatedly Skolemizing and flattening,
Replacing using formula -2,
Keeping (-3 -8 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `phy_mem_unchanged_memory_write_invariant.2`.
Q.E.D.

C.167.10 Phy_Mem_Properties.phy_mem_changed_memory_invariant

Terse proof for `phy_mem_changed_memory_invariant`.

`phy_mem_changed_memory_invariant:`

{1} $\text{changed\_memory\_invariant?}(\text{phy\_mem}, \text{fullset}[\text{Physical\_memory}], \text{in\_memory}(\text{min}, \text{max}))$

Expanding the definition of changed_memory_invariant?,
Rewriting using transformer_invariant_truth, matching in *,
Rewriting using transformer_invariant_truth, matching in *,
Repeatedly Skolemizing and flattening,
Keeping (-3 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `phy_mem_changed_memory_invariant`.
Q.E.D.

C.168 Proofs for Physical_Memory (physical_memory.pvs)

This theory contains no provable formal statements.

C.169 Proofs for Physical_Memory_Corollaries (physical_memory.pvs)

C.169.1 Physical_Memory_Corollaries.physical_read_ok_next_state_TCC1

Terse proof for `physical_read_ok_next_state_TCC1`.

`physical_read_ok_next_state_TCC1:`

{1} ∀ (a: Address, s: Physical_memory[min, max]):
OK?(physical_read(a)(s)) ⊃
OK?[Physical_memory[min, max], Byte](physical_read[min, max](a)(s)) ∨
Exception?[Physical_memory[min, max], Byte](physical_read[min, max](a)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_read_ok_next_state_TCC1`.
Q.E.D.

C.169.2 Physical_Memory_Corollaries.physical_read_ok_next_state

Terse proof for `physical_read_ok_next_state`.

`physical_read_ok_next_state:`

{1} ∀ (a: Address, s: Physical_memory[min, max]):
OK?(physical_read(a)(s)) ⊃ state(physical_read(a)(s)) = s

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_read_ok_next_state`.
Q.E.D.

C.169.3 Physical_Memory_Corollaries.physical_write_ok_next_state_TCC1

Terse proof for `physical_write_ok_next_state_TCC1`.

`physical_write_ok_next_state_TCC1:`

{1} ∀ (a: Address, b: Byte, s: Physical_memory[min, max]):
OK?(physical_write(a, b)(s)) ⊃
OK?[Physical_memory[min, max], Unit](physical_write[min, max](a, b)(s)) ∨
Exception?[Physical_memory[min, max], Unit](physical_write[min, max](a, b)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `physical_write_ok_next_state_TCC1`.
Q.E.D.

C.169.4 Physical_Memory_Corollaries.physical_write_ok_next_state

Terse proof for `physical_write_ok_next_state`.

physical_write_ok_next_state:

{1} ∀ (a: Address, b: Byte, s: Physical_memory[min, max]):
OK?(physical_write(a, b)(s)) ⊃
state(physical_write(a, b)(s)) = s WITH [(a) := b]

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of physical_write_ok_next_state.
Q.E.D.

C.169.5 Physical_Memory_Corollaries.physical_read_ok_get_data

Terse proof for physical_read_ok_get_data.
physical_read_ok_get_data:

{1} ∀ (a: Address, s: Physical_memory[min, max]):
OK?(physical_read(a)(s)) ⊃ data(physical_read(a)(s)) = s(a)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of physical_read_ok_get_data.
Q.E.D.

C.170 Proofs for Physical_Memory_Properties (physical_memory.pvs)

C.170.1 Physical_Memory_Properties.physical_read_ok

Terse proof for physical_read_ok.
physical_read_ok:

{1} ∀ (a: Address, s: Physical_memory[min, max]):
in_memory(min, max)(a) ⊃ OK?(physical_read(a)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of physical_read_ok.
Q.E.D.

C.170.2 Physical_Memory_Properties.physical_write_ok

Terse proof for physical_write_ok.
physical_write_ok:

{1} ∀ (a: Address, b: Byte, s: Physical_memory[min, max]):
in_memory(min, max)(a) ⊃ OK?(physical_write(a, b)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of physical_write_ok.
Q.E.D.

C.171 Proofs for Plain_Mem_Properties (plain_memory.pvs)

C.171.1 Plain_Mem_Properties.plain_mem_write_list_read_list_TCC1

Terse proof for `plain_mem_write_list_read_list_TCC1`.

`plain_mem_write_list_read_list_TCC1`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
       OK?(memory_write_list(pm`mem)(addr, bl)(s)) ∧
        (address_block(addr, size) ⊆ pm`rw_addr) ∧
         pm`states(s) ∧ plain_memory?(pm) ∧ size = length(bl)
        ⊃
        OK?[State, Unit](memory_write_list[State](pm`mem)(addr, bl)(s)) ∨
         Exception?[State, Unit](memory_write_list[State](pm`mem)(addr, bl)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_write_list_read_list_TCC1`.

Q.E.D.

C.171.2 Plain_Mem_Properties.plain_mem_write_list_read_list

Terse proof for `plain_mem_write_list_read_list`.

`plain_mem_write_list_read_list`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
       size = length(bl) ∧
        plain_memory?(pm) ∧
         pm`states(s) ∧
          (address_block(addr, size) ⊆ pm`rw_addr) ∧
           OK?(memory_write_list(pm`mem)(addr, bl)(s)) ∧
            OK?(memory_read_list(pm`mem)(addr, size)
                                 (state(memory_write_list(pm`mem)(addr, bl)(s))))
        ⊃
        data(memory_read_list(pm`mem)(addr, size)
                              (state(memory_write_list(pm`mem)(addr, bl)(s))))
         = bl
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: subset_bigger_union_right

Using lemma unchanged_memory_read_list_write_list,

Replacing using formula -2,

Installing automatic rewrites from: plain_memory_transformers_ok_write_block transformers_ok_union_transformers plain_memory_transformers_ok_write_side_effects_block plain_memory_transformers_ok_read_block plain_memory_transformers_ok_read_side_effects_block plain_memory_side_effect_content_unchanged_read_block plain_memory_side_effect_content_unchanged_write_block plain_memory_changed_block plain_memory_unchanged_block plain_memory_unchanged_invariant

Installing automatic rewrites from: unchanged_memory_invariant_union_transformers plain_memory_unchanged_read_block plain_memory_unchanged_read_side_effect_block plain_memory_unchanged_write_side_effect_block

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_mem_write_list_read_list`.

Q.E.D.

C.171.3 Plain_Mem_Properties.plain_mem_q_read_list_TCC1

Terse proof for `plain_mem_q_read_list_TCC1`.

`plain_mem_q_read_list_TCC1`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory[State], s: State, size: nat,
          q: [State → SuperResult[State]]):
       OK?(q(s)) ∧ OK?(memory_read_list(pm`mem)(addr, size)(s)) ⊃
        OK?[State](q(s)) ∨ abnormal?[State](q(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_q_read_list_TCC1`.

Q.E.D.

C.171.4 Plain_Mem_Properties.plain_mem_q_read_list

Terse proof for `plain_mem_q_read_list`.

`plain_mem_q_read_list`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory[State], s: State, size: nat,
          q: [State → SuperResult[State]]):
       OK?(memory_read_list(pm`mem)(addr, size)(s)) ∧
        OK?(q(s)) ∧
         OK?(memory_read_list(pm`mem)(addr, size)(state(q(s)))) ∧
          plain_memory?(pm) ∧
           unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                                       address_block(addr, size))
            ∧ pm`states(s) ∧ (address_block(addr, size) ⊆ (pm`ro_addr ∪ pm`rw_addr))
       ⊃
        data(memory_read_list(pm`mem)(addr, size)(state(q(s)))) =
         data(memory_read_list(pm`mem)(addr, size)(s))
```

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: union_subset1 subset_reflexive

Using lemma unchanged_memory_read_list[State],

Installing automatic rewrites from: transformers_ok_union_transformers plain_memory_side_effect_content_uncha
plain_memory_transformers_ok_read_block plain_memory_transformers_ok_read_side_effects_block un-
changed_memory_invariant_union_transformers plain_memory_unchanged_read_block plain_memory_unchanged_rea

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_mem_q_read_list`.

Q.E.D.

C.172 Proofs for Plain_Mem_Properties_2 (plain_memory.pvs)

C.172.1 Plain_Mem_Properties_2.plain_memory_inv_pred_write_data_TCC1

Terse proof for `plain_memory_inv_pred_write_data_TCC1`.

`plain_memory_inv_pred_write_data_TCC1:`

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
OK?(write_data(pm, dt)(addr, data)(s)) ∧
(address_block(addr, size(uidt(dt))) ⊆ pm‘rw_addr) ∧
pm‘states(s) ∧ plain_memory?(pm)
⊃
OK?[State, Unit](write_data[State, Data](pm, dt)(addr, data)(s)) ∨
Exception?[State, Unit](write_data[State, Data](pm, dt)(addr, data)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_inv_pred_write_data_TCC1`.
Q.E.D.

C.172.2 Plain_Mem_Properties_2.plain_memory_inv_pred_write_data

The LaTeX code for this proof is broken.

C.172.3 Plain_Mem_Properties_2.plain_memory_read_data_ok

Terse proof for `plain_memory_read_data_ok`.
`plain_memory_read_data_ok:`

{1} ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧
pm‘states(s) ∧
in_blessed_memory?(dt, addr, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
valid_in_mem(pm, dt)(addr)(s)
⊃ OK?(read_data(pm, dt)(addr)(s))

Installing automatic rewrites from: (## read_data)
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Case splitting on OK?(memory_read_list(pm!1‘mem)(addr!1, size(uidt(dt!1)))(s!1)),
we get 2 subgoals:
`plain_memory_read_data_ok.1:`

{-1} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))
{-2} interpreted_data_type?[Data](dt′)
{-3} plain_memory?(pm′)
{-4} pm′‘states(s′)
{-5} in_blessed_memory?(dt′, addr′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6} valid_in_mem(pm′, dt′)(addr′)(s′)

{1} OK?(CASES memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′) OF
OK(state, data): ok_lift(from_byte(dt′)(data, addr′))(state),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)

Simplifying, rewriting, and recording with decision procedures,
Rewriting using ok_lift_ok, matching in *,

we get 2 subgoals:
`plain_memory_read_data_ok.1.1:`

```
{-1}  OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))
{-2}  interpreted_data_type?[Data](dt′)
{-3}  plain_memory?(pm′)
{-4}  pm′‘states(s′)
{-5}  in_blessed_memory?(dt′, addr′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6}  valid_in_mem(pm′, dt′)(addr′)(s′)
------------------------------------------------------------------------
{1}   OK?(OK(state(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′)),
                down(from_byte(dt′)
                              (data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′)),
                                addr′))))
```

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_read_data_ok.1.1`.
`plain_memory_read_data_ok.1.2:`

```
{-1}  OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))
{-2}  interpreted_data_type?[Data](dt′)
{-3}  plain_memory?(pm′)
{-4}  pm′‘states(s′)
{-5}  in_blessed_memory?(dt′, addr′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-6}  valid_in_mem(pm′, dt′)(addr′)(s′)
------------------------------------------------------------------------
{1}   up?(from_byte(dt′)
                (data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′)), addr′))
{2}   OK?(ok_lift(from_byte(dt′)
                           (data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′)),
                             addr′))
                  (state(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))))
```

Rewriting using valid_from_byte, matching in *,
Expanding the definition of valid_in_mem,
Expanding the definition of valid_in_mem,
which is trivially true.
This completes the proof of `plain_memory_read_data_ok.1.2`.
`plain_memory_read_data_ok.2:`

```
{-1}  interpreted_data_type?[Data](dt′)
{-2}  plain_memory?(pm′)
{-3}  pm′‘states(s′)
{-4}  in_blessed_memory?(dt′, addr′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}  valid_in_mem(pm′, dt′)(addr′)(s′)
------------------------------------------------------------------------
{1}   OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))
{2}   OK?(CASES memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′) OF
            OK(state, data): ok_lift(from_byte(dt′)(data, addr′))(state),
            Exception(ex_type, state): Exception(ex_type, state),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Expanding the definition of in_blessed_memory?,
Rewriting using memory_read_list_ok, matching in * where states gets pm!1‘states,
we get 2 subgoals:

`plain_memory_read_data_ok.2.1:`

{-1} interpreted_data_type?[Data](dt′)
{-2} plain_memory?(pm′)
{-3} pm′`states(s′)
{-4} (address_block(addr′, size(uidt(dt′))) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-5} valid_in_mem(pm′, dt′)(addr′)(s′)

{1} transformer_invariant?(pm′`states,
(memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ m
{2} OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′))
{3} OK?(CASES memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′) OF
OK(state, data): ok_lift(from_byte(dt′)(data, addr′))(state),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)

Rewriting using plain_memory_transformer_invariant_read_list_block, matching in * where pm gets pm!1,
This completes the proof of `plain_memory_read_data_ok.2.1`.
`plain_memory_read_data_ok.2.2:`

{-1} interpreted_data_type?[Data](dt′)
{-2} plain_memory?(pm′)
{-3} pm′`states(s′)
{-4} (address_block(addr′, size(uidt(dt′))) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-5} valid_in_mem(pm′, dt′)(addr′)(s′)

{1} transformers_ok?(pm′`states,
(memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ memory_
{2} OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′))
{3} OK?(CASES memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′) OF
OK(state, data): ok_lift(from_byte(dt′)(data, addr′))(state),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)

Rewriting using plain_memory_transformers_ok_read_list_block, matching in * where pm gets pm!1,
This completes the proof of `plain_memory_read_data_ok.2.2`.
Q.E.D.

C.172.4 Plain_Mem_Properties_2.plain_memory_write_data_ok

Terse proof for `plain_memory_write_data_ok`.
`plain_memory_write_data_ok:`

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State],
s: State):
plain_memory?(pm) ∧ pm`states(s) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ⊃
OK?(write_data(pm, dt)(addr, data)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of write_data,
Expanding the definition of write_data,

Installing automatic rewrites from: length_to_byte

Expanding the definition of in_blessed_memory?,

Rewriting using memory_write_list_ok, matching in * where pm gets pm!1‘mem, states gets pm!1‘states,

we get 3 subgoals:

`plain_memory_write_data_ok.1:`

```
{-1}  interpreted_data_type?[Data](dt')
{-2}  plain_memory?(pm')
{-3}  pm'‘states(s')
{-4}  (address_block(addr', size(uidt(dt'))) ⊆ pm'‘rw_addr)
------------------------------------------------------------------------------
{1}   transformer_invariant?(pm'‘states,
                             (memory_write_transformers(pm'‘mem, address_block(addr', size(uid
{2}   OK?(memory_write_list(pm'‘mem)(addr', to_byte(dt')(data', addr'))(s'))
```

Using lemma plain_memory_transformer_invariant_write_list_block[State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_ok.1`.

`plain_memory_write_data_ok.2:`

```
{-1}  interpreted_data_type?[Data](dt')
{-2}  plain_memory?(pm')
{-3}  pm'‘states(s')
{-4}  (address_block(addr', size(uidt(dt'))) ⊆ pm'‘rw_addr)
------------------------------------------------------------------------------
{1}   transformers_ok?(pm'‘states,
                       (memory_write_transformers(pm'‘mem, address_block(addr', size(uidt(dt'))))
{2}   OK?(memory_write_list(pm'‘mem)(addr', to_byte(dt')(data', addr'))(s'))
```

Using lemma plain_memory_transformers_ok_write_list_block[State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_ok.2`.

`plain_memory_write_data_ok.3:`

```
{-1}  interpreted_data_type?[Data](dt')
{-2}  plain_memory?(pm')
{-3}  pm'‘states(s')
{-4}  (address_block(addr', size(uidt(dt'))) ⊆ pm'‘rw_addr)
------------------------------------------------------------------------------
{1}   side_effect_content_unchanged(address_block(addr', size(uidt(dt'))), pm'‘states,
                                    memory_write_side_effect(pm'‘mem))
{2}   OK?(memory_write_list(pm'‘mem)(addr', to_byte(dt')(data', addr'))(s'))
```

Using lemma plain_memory_side_effect_content_unchanged_write_block[State],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_ok.3`.

Q.E.D.

C.172.5 Plain_Mem_Properties_2.plain_memory_write_data_valid_TCC1

Terse proof for `plain_memory_write_data_valid_TCC1`.

`plain_memory_write_data_valid_TCC1`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm`states(s) ∧ plain_memory?(pm) ⊃
OK?[State, Unit](write_data[State, Data](pm, dt)(addr, data)(s)) ∨
Exception?[State, Unit](write_data[State, Data](pm, dt)(addr, data)(s))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_write_data_ok,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_write_data_valid_TCC1`.

Q.E.D.

C.172.6 Plain_Mem_Properties_2.plain_memory_write_data_valid

Terse proof for `plain_memory_write_data_valid`.

`plain_memory_write_data_valid`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧ pm`states(s) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ⊃
valid_in_mem(pm, dt)(addr)(state(write_data(pm, dt)(addr, data)(s)))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_write_data_ok,

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of valid_in_mem,

Expanding the definition of valid_in_mem,

Expanding the definition of write_data,

Expanding the definition of write_data,

Installing automatic rewrites from: length_to_byte valid_to_byte subset_bigger_union_right

Using lemma plain_mem_write_list_read_list[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_write_data_valid.1:`

{-1} plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} OK?(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′))(s′))
{-4} data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data′, addr′))
(s′))))
= to_byte(dt′)(data′, addr′)
{-5} interpreted_data_type?[Data](dt′)
{-6} in_blessed_memory?(dt′, addr′, pm′‘rw_addr)
{-7} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data′, addr′))
(s′))))

{1} dt′‘uidt‘valid?
(data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data′, addr′)
(s′)))),
addr′)

Replacing using formula -4,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_write_data_valid.1`.
`plain_memory_write_data_valid.2:`

{-1} plain_memory?(pm′)
{-2} pm′‘states(s′)
{-3} OK?(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′))(s′))
{-4} interpreted_data_type?[Data](dt′)
{-5} in_blessed_memory?(dt′, addr′, pm′‘rw_addr)
{-6} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data′, addr′))
(s′))))

{1} (address_block(addr′, size(uidt(dt′))) ⊆ pm′‘rw_addr)
{2} dt′‘uidt‘valid?
(data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data′, addr′)
(s′)))),
addr′)

Expanding the definition of in_blessed_memory?,
which is trivially true.
This completes the proof of `plain_memory_write_data_valid.2`.
Q.E.D.

C.172.7 Plain_Mem_Properties_2.plain_memory_transformers_ok_write_data

Terse proof for `plain_memory_transformers_ok_write_data`.

plain_memory_transformers_ok_write_data:

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]),
          pm: Plain_Memory[State]):
        plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ⊃
         transformers_ok?(pm`states, singleton(expr_2_super(write_data(pm, dt)(addr, data))))
```

Expanding the definition of transformers_ok?,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: length_to_byte singleton write_data plain_memory_side_effect_content_unchanged_write_b expr_2_super expr_2_super_res

Simplifying, rewriting, and recording with decision procedures,

Replacing using formula -5,

Using lemma memory_write_list_ok,

Using lemma plain_memory_transformer_invariant_write_list_block[State],

Using lemma plain_memory_transformers_ok_write_list_block[State],

Expanding the definition of in_blessed_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_transformers_ok_write_data.

Q.E.D.

C.172.8 Plain_Mem_Properties_2.plain_memory_transformer_invariant_write_data

Terse proof for plain_memory_transformer_invariant_write_data.

plain_memory_transformer_invariant_write_data:

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]),
          pm: Plain_Memory[State]):
        plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ⊃
         transformer_invariant?(pm`states,
                                singleton(expr_2_super(write_data(pm, dt)(addr, data))))
```

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Expanding the definition of write_data,

Expanding the definition of write_data,

Using lemma transformer_invariant_write_list,

Using lemma plain_memory_transformer_invariant_write_list_block[State],

Installing automatic rewrites from: length_to_byte

Expanding the definition of in_blessed_memory?,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_transformer_invariant_write_data.1:`

{-1}	plain_memory?(pm′)
{-2}	transformer_invariant?(pm′‘states, (memory_write_transformers(pm′‘mem, address_block(addr′, size(uid
{-3}	pm′‘states(s')
{-4}	interpreted_data_type?[Data](dt′)
{-5}	(address_block(addr′, size(uidt(dt′))) ⊆ pm′‘rw_addr)
{-6}	q' = expr_2_super(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′)))
{1}	side_effect_content_unchanged(address_block(addr′, size(uidt(dt′))), pm′‘states, memory_write_side_effect(pm′‘mem))
{2}	result_pred(pm′‘states)(q'(s'))

Rewriting using plain_memory_side_effect_content_unchanged_write_block, matching in *,
This completes the proof of `plain_memory_transformer_invariant_write_data.1`.
`plain_memory_transformer_invariant_write_data.2:`

{-1}	plain_memory?(pm′)
{-2}	transformer_invariant?(pm′‘states, (memory_write_transformers(pm′‘mem, address_block(addr′, size(uid
{-3}	pm′‘states(s')
{-4}	interpreted_data_type?[Data](dt′)
{-5}	(address_block(addr′, size(uidt(dt′))) ⊆ pm′‘rw_addr)
{-6}	q' = expr_2_super(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′)))
{1}	transformer_invariant?(pm′‘states, (memory_write_transformers(pm′‘mem, address_block(addr′, size(uid
{2}	result_pred(pm′‘states)(q'(s'))

Keeping (-2 -3 -5 1) and hiding *,
Rewriting using add_as_union, matching in *,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Hiding formulas: (-1 2),
Expanding the definition of subset?,
Expanding the definition of union,
Expanding the definition of member,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of memory_write_side_effect_super_transformers,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_reflexive, matching in *,
Expanding the definition of singleton,
which is trivially true.
This completes the proof of `plain_memory_transformer_invariant_write_data.2`.
Q.E.D.

C.172.9 Plain_Mem_Properties_2.plain_memory_transformer_invariant_read_data

Terse proof for `plain_memory_transformer_invariant_read_data`.

`plain_memory_transformer_invariant_read_data:`

```
{1}  ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
         transformer_invariant?(pm‘states, singleton(expr_2_super(read_data(pm, dt)(addr))))
```

Expanding the definition of transformer_invariant?,
Installing automatic rewrites from: result_pred has_next_state
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of singleton,
Replacing using formula -5,
Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ok_lift! read_data! ##!)
Simplifying, rewriting, and recording with decision procedures,
Using lemma memory_read_list_next_ok[State],
Expanding the definition of in_blessed_memory?,
Rewriting using plain_memory_transformers_ok_read_list_block[State], matching in *,
Rewriting using plain_memory_transformer_invariant_read_list_block[State], matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_transformer_invariant_read_data`.
Q.E.D.

C.172.10 Plain_Mem_Properties_2.plain_memory_unchanged_memory_invariant_write_data

Terse proof for `plain_memory_unchanged_memory_invariant_write_data`.

`plain_memory_unchanged_memory_invariant_write_data:`

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]),
          pm: Plain_Memory[State]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm‘rw_addr) ⊃
         unchanged_memory_invariant?(pm‘mem, pm‘states,
                                     singleton(expr_2_super(write_data(pm, dt)(addr, data))),
                                     ((pm‘ro_addr ∪ pm‘rw_addr) \ address_block(addr, size(uidt(dt)))))
```

Expanding the definition of unchanged_memory_invariant?,
Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`plain_memory_unchanged_memory_invariant_write_data.1:`

```
{-1}  interpreted_data_type?[Data](dt′)
{-2}  plain_memory?(pm′)
{-3}  in_blessed_memory?(dt′, addr′, pm′‘rw_addr)
{1}   transformer_invariant?(pm′‘states,
                             singleton(expr_2_super(write_data(pm′, dt′)(addr′, data′))))
```

Using lemma plain_memory_transformer_invariant_write_data,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.1`.

`plain_memory_unchanged_memory_invariant_write_data.2:`

{-1} interpreted_data_type?[Data](dt′)
{-2} plain_memory?(pm′)
{-3} in_blessed_memory?(dt′, addr′, pm′‘rw_addr)

{1} ∀ (s: State, q: [State → SuperResult[State]], a: Address):
pm′‘states(s) ∧
singleton(expr_2_super(write_data(pm′, dt′)(addr′, data′)))(q) ∧
difference((pm′‘ro_addr ∪ pm′‘rw_addr), address_block(addr′, size(uidt(dt′))))
(a)
∧
OK?(q(s)) ∧
OK?(memory_read(pm′‘mem)(a)(s)) ∧
OK?(memory_read(pm′‘mem)(a)(state(q(s))))
⊃
data(memory_read(pm′‘mem)(a)(state(q(s)))) =
data(memory_read(pm′‘mem)(a)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of write_data,
Expanding the definition of write_data,
Installing automatic rewrites from: length_to_byte ok_expr_2_super state_expr_2_super has_next_state subset_singleton
Expanding the definition of singleton,
Replacing using formula -2,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of in_blessed_memory?,
Using lemma unchanged_memory_invariant_unchanged_write_list[State],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 4 subgoals:
`plain_memory_unchanged_memory_invariant_write_data.2.1:`

{-1} pm′‘states(s′)
{-2} q′ = expr_2_super(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′)))
{-3} difference((pm′‘ro_addr ∪ pm′‘rw_addr), address_block(addr′, size(uidt(dt′))))
(a′)
{-4} OK?(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′))(s′))
{-5} OK?(memory_read(pm′‘mem)(a′)(s′))
{-6} OK?(memory_read(pm′‘mem)
(a′)
(state(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′))
(s′))))
{-7} interpreted_data_type?[Data](dt′)
{-8} plain_memory?(pm′)
{-9} (address_block(addr′, size(uidt(dt′))) ⊆ pm′‘rw_addr)

{1} side_effect_content_unchanged(address_block(addr′, size(uidt(dt′))), pm′‘states,
memory_write_side_effect(pm′‘mem))
{2} data(memory_read(pm′‘mem)
(a′)
(state(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data′, addr′))
(s′))))
= data(memory_read(pm′‘mem)(a′)(s′))

Rewriting using plain_memory_side_effect_content_unchanged_write_block, matching in *,
This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.1`.

`plain_memory_unchanged_memory_invariant_write_data.2.2`:

```
{-1}  pm'`states(s')
{-2}  q' = expr_2_super(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr')))
{-3}  difference((pm'`ro_addr ∪ pm'`rw_addr), address_block(addr', size(uidt(dt'))))
                 (a')
{-4}  OK?(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr'))(s'))
{-5}  OK?(memory_read(pm'`mem)(a')(s'))
{-6}  OK?(memory_read(pm'`mem)
                 (a')
                 (state(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr'))
                                          (s'))))
{-7}  interpreted_data_type?[Data](dt')
{-8}  plain_memory?(pm')
{-9}  (address_block(addr', size(uidt(dt'))) ⊆ pm'`rw_addr)
-------------------------------------------------------------------------------
 {1}  transformers_ok?(pm'`states, singleton(expr_2_super(memory_read(pm'`mem)(a'))))
 {2}  data(memory_read(pm'`mem)
                 (a')
                 (state(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr'))
                                          (s'))))
        = data(memory_read(pm'`mem)(a')(s'))
```

Keeping (-3 -8 1) and hiding *,
Using lemma plain_memory_transformers_ok_read_ro_rw,
Simplifying, rewriting, and recording with decision procedures,
Using lemma transformers_ok_mono_transformers[State],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using memory_read_transformers_memory_read, matching in *,
Keeping (-2 1) and hiding *,
Expanding the definition(s) of (difference member),
This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.2`.
`plain_memory_unchanged_memory_invariant_write_data.2.3`:

```
{-1}  pm'`states(s')
{-2}  q' = expr_2_super(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr')))
{-3}  difference((pm'`ro_addr ∪ pm'`rw_addr), address_block(addr', size(uidt(dt'))))
                 (a')
{-4}  OK?(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr'))(s'))
{-5}  OK?(memory_read(pm'`mem)(a')(s'))
{-6}  OK?(memory_read(pm'`mem)
                 (a')
                 (state(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr'))
                                          (s'))))
{-7}  interpreted_data_type?[Data](dt')
{-8}  plain_memory?(pm')
{-9}  (address_block(addr', size(uidt(dt'))) ⊆ pm'`rw_addr)
-------------------------------------------------------------------------------
 {1}  transformers_ok?(pm'`states,
                       (memory_write_transformers(pm'`mem, address_block(addr', size(uidt(dt')))) ∪ memory
 {2}  data(memory_read(pm'`mem)
                 (a')
                 (state(memory_write_list(pm'`mem)(addr', to_byte(dt')(data', addr'))
                                          (s'))))
        = data(memory_read(pm'`mem)(a')(s'))
```

Rewriting using plain_memory_transformers_ok_write_list_block, matching in *,

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.3`.

`plain_memory_unchanged_memory_invariant_write_data.2.4:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{pm'`states}(s') \\
\{-2\} & q' = \mathrm{expr\_2\_super}(\mathrm{memory\_write\_list}(\mathrm{pm'`mem})(\mathrm{addr'}, \mathrm{to\_byte}(\mathrm{dt'})(\mathrm{data'}, \mathrm{addr'}))) \\
\{-3\} & \mathrm{difference}((\mathrm{pm'`ro\_addr} \cup \mathrm{pm'`rw\_addr}), \mathrm{address\_block}(\mathrm{addr'}, \mathrm{size}(\mathrm{uidt}(\mathrm{dt'})))) \\
 & \qquad (a') \\
\{-4\} & \mathrm{OK?}(\mathrm{memory\_write\_list}(\mathrm{pm'`mem})(\mathrm{addr'}, \mathrm{to\_byte}(\mathrm{dt'})(\mathrm{data'}, \mathrm{addr'}))(s')) \\
\{-5\} & \mathrm{OK?}(\mathrm{memory\_read}(\mathrm{pm'`mem})(a')(s')) \\
\{-6\} & \mathrm{OK?}(\mathrm{memory\_read}(\mathrm{pm'`mem}) \\
 & \qquad (a') \\
 & \qquad (\mathrm{state}(\mathrm{memory\_write\_list}(\mathrm{pm'`mem})(\mathrm{addr'}, \mathrm{to\_byte}(\mathrm{dt'})(\mathrm{data'}, \mathrm{addr'})) \\
 & \qquad\qquad (s')))) \\
\{-7\} & \mathrm{interpreted\_data\_type?}[\mathrm{Data}](\mathrm{dt'}) \\
\{-8\} & \mathrm{plain\_memory?}(\mathrm{pm'}) \\
\{-9\} & (\mathrm{address\_block}(\mathrm{addr'}, \mathrm{size}(\mathrm{uidt}(\mathrm{dt'}))) \subseteq \mathrm{pm'`rw\_addr}) \\
\hline
\{1\} & \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm'`mem}, \mathrm{pm'`states}, \\
 & \qquad (\mathrm{memory\_write\_transformers}(\mathrm{pm'`mem}, \mathrm{address\_block}(\mathrm{addr'}, \mathrm{si} \\
 & \qquad \mathrm{singleton}(a')) \\
\{2\} & \mathrm{data}(\mathrm{memory\_read}(\mathrm{pm'`mem}) \\
 & \qquad (a') \\
 & \qquad (\mathrm{state}(\mathrm{memory\_write\_list}(\mathrm{pm'`mem})(\mathrm{addr'}, \mathrm{to\_byte}(\mathrm{dt'})(\mathrm{data'}, \mathrm{addr'})) \\
 & \qquad\qquad (s')))) \\
 & = \mathrm{data}(\mathrm{memory\_read}(\mathrm{pm'`mem})(a')(s'))
\end{array}
$$

Keeping (-3 -8 -10 1) and hiding *,

Rewriting using unchanged_memory_invariant_union_transformers[State], matching in *,

we get 2 subgoals:

`plain_memory_unchanged_memory_invariant_write_data.2.4.1:`

$$
\begin{array}{ll}
\{-1\} & \mathrm{difference}((\mathrm{pm'`ro\_addr} \cup \mathrm{pm'`rw\_addr}), \mathrm{address\_block}(\mathrm{addr'}, \mathrm{size}(\mathrm{uidt}(\mathrm{dt'})))) \\
 & \qquad (a') \\
\{-2\} & \mathrm{plain\_memory?}(\mathrm{pm'}) \\
\hline
\{1\} & \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm'`mem}, \mathrm{pm'`states}, \\
 & \qquad \mathrm{memory\_write\_transformers}(\mathrm{pm'`mem}, \\
 & \qquad\qquad \mathrm{address\_block}(\mathrm{addr'}, \\
 & \qquad\qquad\qquad \mathrm{size}(\mathrm{uidt}(\mathrm{dt'})) \\
 & \qquad \mathrm{singleton}(a')) \\
\{2\} & \mathrm{unchanged\_memory\_invariant?}(\mathrm{pm'`mem}, \mathrm{pm'`states}, \\
 & \qquad (\mathrm{memory\_write\_transformers}(\mathrm{pm'`mem}, \mathrm{address\_block}(\mathrm{addr'}, \mathrm{si} \\
 & \qquad \mathrm{singleton}(a'))
\end{array}
$$

Hiding formulas: 2,

Using lemma unchanged_memory_invariant_union[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 3 subgoals:

`plain_memory_unchanged_memory_invariant_write_data.2.4.1.1`:

```
{-1}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem,
                                                            address_block(addr',
                                                                          size(uidt(dt')))),
                                  ((pm'`ro_addr ∪ pm'`rw_addr) \ address_block(addr', size(uidt(dt')))))
{-2}  difference((pm'`ro_addr ∪ pm'`rw_addr), address_block(addr', size(uidt(dt'))))
                 (a')
{-3}  plain_memory?(pm')
-------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem,
                                                            address_block(addr',
                                                                          size(uidt(dt')))),
                                  singleton(a'))
```

Using lemma unchanged_memory_invariant_mono[State],

Installing automatic rewrites from: subset_equal

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.4.1.1`.

`plain_memory_unchanged_memory_invariant_write_data.2.4.1.2`:

```
{-1}  difference((pm'`ro_addr ∪ pm'`rw_addr), address_block(addr', size(uidt(dt'))))
                 (a')
{-2}  plain_memory?(pm')
-------------------------------------------------------------------------------
 {1}  unchanged_memory_write_invariant?(pm'`mem, pm'`states, pm'`rw_addr)
 {2}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem,
                                                            address_block(addr',
                                                                          size(uidt(dt')))),
                                  singleton(a'))
```

Rewriting using plain_memory_unchanged_memory_write_invariant, matching in *,

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.4.1.2`.

`plain_memory_unchanged_memory_invariant_write_data.2.4.1.3`:

```
{-1}  difference((pm'`ro_addr ∪ pm'`rw_addr), address_block(addr', size(uidt(dt'))))
                 (a')
{-2}  plain_memory?(pm')
-------------------------------------------------------------------------------
 {1}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem, pm'`rw_addr), pm'`ro_addr)
 {2}  unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  memory_write_transformers(pm'`mem,
                                                            address_block(addr',
                                                                          size(uidt(dt')))),
                                  singleton(a'))
```

Rewriting using plain_memory_unchanged_memory_invariant_write, matching in *,

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.4.1.3`.

`plain_memory_unchanged_memory_invariant_write_data.2.4.2:`

{-1} difference((pm′‘ro_addr ∪ pm′‘rw_addr), address_block(addr′, size(uidt(dt′)))) (a′)
{-2} plain_memory?(pm′)

{1} unchanged_memory_invariant?(pm′‘mem, pm′‘states, memory_write_side_effect_super_transformers(pm′‘mem, address_block (addr′, size (uidt(dt singleton(a′))
{2} unchanged_memory_invariant?(pm′‘mem, pm′‘states, (memory_write_transformers(pm′‘mem, address_block(addr′, si singleton(a′))

Hiding formulas: 2,

Using lemma plain_memory_unchanged_invariant,

Using lemma unchanged_memory_invariant_mono[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_unchanged_memory_invariant_write_data.2.4.2.1:`

{-1} unchanged_memory_invariant?(pm′‘mem, pm′‘states, ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pn (pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} plain_memory?(pm′)
{-3} difference((pm′‘ro_addr ∪ pm′‘rw_addr), address_block(addr′, size(uidt(dt′)))) (a′)

{1} union(pm′‘ro_addr, pm′‘rw_addr)(a′)
{2} unchanged_memory_invariant?(pm′‘mem, pm′‘states, memory_write_side_effect_super_transformers(pm′‘mem, address_block (addr′, size (uidt(dt singleton(a′))

Keeping (-3 1) and hiding *,

Expanding the definition(s) of (difference member),

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.4.2.1`.

`plain_memory_unchanged_memory_invariant_write_data.2.4.2.2:`

```
{-1}  unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  ((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪
                                  (pm′‘ro_addr ∪ pm′‘rw_addr))
{-2}  plain_memory?(pm′)
{-3}  difference((pm′‘ro_addr ∪ pm′‘rw_addr), address_block(addr′, size(uidt(dt′))))
                (a′)
------------------------------------------------------------------------------------------------------
{1}   (memory_write_side_effect_super_transformers(pm′‘mem, address_block(addr′, size(uidt(dt′))))) ⊆ ((pm′‘other
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states,
                                  memory_write_side_effect_super_transformers(pm′‘mem,
                                                                              address_block
                                                                              (addr′,
                                                                                 size
                                                                                 (uidt(dt′)))),
                                  singleton(a′))
```

Keeping (-4 1) and hiding *,

Installing automatic rewrites from: memory_write_side_effect_super_transformers_mono

Rewriting using subset_bigger_union_right, matching in *,

Rewriting using subset_bigger_union_right, matching in *,

This completes the proof of `plain_memory_unchanged_memory_invariant_write_data.2.4.2.2`.

Q.E.D.

C.172.11 Plain_Mem_Properties_2.plain_memory_unchanged_memory_invariant_read_data

Terse proof for `plain_memory_unchanged_memory_invariant_read_data`.

`plain_memory_unchanged_memory_invariant_read_data:`

```
------------------------------------------------------------------------------
{1}   ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]):
        plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
          unchanged_memory_invariant?(pm‘mem, pm‘states,
                                      singleton(expr_2_super(read_data(pm, dt)(addr))),
                                      (pm‘ro_addr ∪ pm‘rw_addr))
```

Installing automatic rewrites from: ok_expr_2_super state_expr_2_super singleton ## ok_lift result_pred has_next_state has_next_state_expr_2_super in_blessed_memory_subset transformers_ok_union_transformers plain_memory_transformers_ok_read_side_effects_ro_rw plain_memory_transformers_ok_read_ro_rw plain_memory_transformers_ok_read_side_effects_block plain_memory_transformers_ok_read_block

Repeatedly Skolemizing and flattening,

Using lemma unchanged_memory_invariant_read_list[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_unchanged_memory_invariant_read_data.1:`

```
{-1}  unchanged_memory_invariant?(pm'‘mem, pm'‘states,
                                  singleton(expr_2_super(memory_read_list(pm'‘mem)
                                                                          (addr',
                                                                            size(uidt(dt'
                                  (pm'‘ro_addr ∪ pm'‘rw_addr))
{-2}  interpreted_data_type?[Data](dt')
{-3}  plain_memory?(pm')
{-4}  in_blessed_memory?(dt', addr', (pm'‘ro_addr ∪ pm'‘rw_addr))
------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'‘mem, pm'‘states,
                                  singleton(expr_2_super(read_data(pm', dt')(addr'))),
                                  (pm'‘ro_addr ∪ pm'‘rw_addr))
```

Expanding the definition of unchanged_memory_invariant?,
Applying propositional simplification,
we get 2 subgoals:
`plain_memory_unchanged_memory_invariant_read_data.1.1:`

```
{-1}  unchanged_memory_invariant?(pm'‘mem, pm'‘states,
                                  singleton(expr_2_super(memory_read_list(pm'‘mem)
                                                                          (addr',
                                                                            size(uidt(dt'
                                  (pm'‘ro_addr ∪ pm'‘rw_addr))
{-2}  interpreted_data_type?[Data](dt')
{-3}  plain_memory?(pm')
{-4}  in_blessed_memory?(dt', addr', (pm'‘ro_addr ∪ pm'‘rw_addr))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm'‘states,
                             singleton(expr_2_super(read_data(pm', dt')(addr'))))
```

Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma memory_read_list_ok[State],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`plain_memory_unchanged_memory_invariant_read_data.1.1.1:`

```
{-1}  pm'‘states(s')
{-2}  OK?(memory_read_list(pm'‘mem)(addr', size(uidt(dt')))(s'))
{-3}  q' = expr_2_super(read_data(pm', dt')(addr'))
{-4}  has_next_state(q'(s'))
{-5}  unchanged_memory_invariant?(pm'‘mem, pm'‘states,
                                  singleton(expr_2_super(memory_read_list(pm'‘mem)
                                                                          (addr',
                                                                            size(uidt(dt'
                                  (pm'‘ro_addr ∪ pm'‘rw_addr))
{-6}  interpreted_data_type?[Data](dt')
{-7}  plain_memory?(pm')
{-8}  in_blessed_memory?(dt', addr', (pm'‘ro_addr ∪ pm'‘rw_addr))
------------------------------------------------------------------------
{1}   pm'‘states(state(q'(s')))
```

Replacing using formula -3,
Hiding formulas: -3,
Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: read_data ok_lift!
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma expr_transformer_invariant_next_ok,
Forward chaining on unchanged_memory_invariant_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_unchanged_memory_invariant_read_data.1.1.1`.
`plain_memory_unchanged_memory_invariant_read_data.1.1.2`:

```
{-1}  pm′`states(s′)
{-2}  q′ = expr_2_super(read_data(pm′, dt′)(addr′))
{-3}  has_next_state(q′(s′))
{-4}  unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(memory_read_list(pm′`mem)
                                                         (addr′,
                                                          size(uidt(dt′))))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{-5}  interpreted_data_type?[Data](dt′)
{-6}  plain_memory?(pm′)
{-7}  in_blessed_memory?(dt′, addr′, (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   transformer_invariant?(pm′`states,
                             (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ m
{2}   pm′`states(state(q′(s′)))
```

Rewriting using plain_memory_transformer_invariant_read_list_block, matching in *,
This completes the proof of `plain_memory_unchanged_memory_invariant_read_data.1.1.2`.
`plain_memory_unchanged_memory_invariant_read_data.1.2`:

```
{-1}  unchanged_memory_invariant?(pm′`mem, pm′`states,
                                  singleton(expr_2_super(memory_read_list(pm′`mem)
                                                         (addr′,
                                                          size(uidt(dt′))))),
                                  (pm′`ro_addr ∪ pm′`rw_addr))
{-2}  interpreted_data_type?[Data](dt′)
{-3}  plain_memory?(pm′)
{-4}  in_blessed_memory?(dt′, addr′, (pm′`ro_addr ∪ pm′`rw_addr))
------------------------------------------------------------------------
{1}   ∀ (s: State, q: [State → SuperResult[State]], a: Address):
        pm′`states(s) ∧
         singleton(expr_2_super(read_data(pm′, dt′)(addr′)))(q) ∧
          union(pm′`ro_addr, pm′`rw_addr)(a) ∧
           OK?(q(s)) ∧
            OK?(memory_read(pm′`mem)(a)(s)) ∧
             OK?(memory_read(pm′`mem)(a)(state(q(s))))
         ⊃
         data(memory_read(pm′`mem)(a)(state(q(s)))) =
          data(memory_read(pm′`mem)(a)(s))
```

Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -2,
Hiding formulas: -2,
Installing automatic rewrites from: read_data ok_lift! ##!
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma expr_unchanged_memory_invariant_unchanged[State, list[Byte]],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_unchanged_memory_invariant_read_data.1.2`.
`plain_memory_unchanged_memory_invariant_read_data.2`:

```
{-1}  interpreted_data_type?[Data](dt')
{-2}  plain_memory?(pm')
{-3}  in_blessed_memory?(dt', addr', (pm'`ro_addr ∪ pm'`rw_addr))
------------------------------------------------------------------------
{1}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  (memory_read_transformers(pm'`mem, address_block(addr', siz
                                  (pm'`ro_addr ∪ pm'`rw_addr))
{2}   unchanged_memory_invariant?(pm'`mem, pm'`states,
                                  singleton(expr_2_super(read_data(pm', dt')(addr'))),
                                  (pm'`ro_addr ∪ pm'`rw_addr))
```

Hiding formulas: 2,
Installing automatic rewrites from: plain_memory_unchanged_read_block_ro_rw_addr plain_memory_unchanged_r
Rewriting using unchanged_memory_invariant_union_transformers, matching in *,
This completes the proof of `plain_memory_unchanged_memory_invariant_read_data.2`.
Q.E.D.

C.172.12 Plain_Mem_Properties_2.plain_memory_read_data_q_ok_TCC1

Terse proof for `plain_memory_read_data_q_ok_TCC1`.
`plain_memory_read_data_q_ok_TCC1`:

```
------------------------------------------------------------------------
{1}   ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
          q: [State → SuperResult[State]]):
        plain_memory?(pm) ∧
         in_blessed_memory?(dt, addr, (pm`ro_addr ∪ pm`rw_addr)) ∧
          unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                                      address_block(addr, size(uidt(dt))))
            ∧ pm`states(s) ∧ OK?(q(s)) ∧ valid_in_mem(pm, dt)(addr)(s)
         ⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_read_data_q_ok_TCC1`.
Q.E.D.

C.172.13 Plain_Mem_Properties_2.plain_memory_read_data_q_ok

Terse proof for `plain_memory_read_data_q_ok`.
`plain_memory_read_data_q_ok`:

```
------------------------------------------------------------------------
{1}   ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
          q: [State → SuperResult[State]]):
        plain_memory?(pm) ∧
         in_blessed_memory?(dt, addr, (pm`ro_addr ∪ pm`rw_addr)) ∧
          unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
                                      address_block(addr, size(uidt(dt))))
            ∧ pm`states(s) ∧ OK?(q(s)) ∧ valid_in_mem(pm, dt)(addr)(s)
         ⊃ OK?(read_data(pm, dt)(addr)(state(q(s))))
```

Installing automatic rewrites from: (valid_in_mem read_data ## expr_2_super expr_2_super_res ok_lift has_next_state)

Repeatedly Skolemizing and flattening,

Using lemma plain_mem_q_read_list[State],

Using lemma memory_read_list_ok[State],

Using lemma memory_read_list_ok[State],

Expanding the definition of in_blessed_memory?,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_transformers_ok_read_list_block[State],

Using lemma plain_memory_transformer_invariant_read_list_block[State],

Case splitting on OK?(memory_read_list(pm!1`mem)(addr!1, size(uidt(dt!1))) (state(q!1(s!1)))),

we get 2 subgoals:

`plain_memory_read_data_q_ok.1:`

{-1} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(state(q′(s′))))
{-2} plain_memory?(pm′) ∧
(address_block(addr′, size(uidt(dt′))) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
⊃
transformer_invariant?(pm′‘states,
(memory_read_transformers(pm′‘mem, address_block(addr′, size(ui
{-3} plain_memory?(pm′) ∧
(address_block(addr′, size(uidt(dt′))) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
⊃
transformers_ok?(pm′‘states,
(memory_read_transformers(pm′‘mem, address_block(addr′, size(uidt(dt′)
{-4} transformer_invariant?(pm′‘states,
(memory_read_transformers(pm′‘mem, address_block(addr′, size(uid
∧
transformers_ok?(pm′‘states,
(memory_read_transformers(pm′‘mem, address_block(addr′, size(uidt(dt′)
∧ pm′‘states(state(q′(s′)))
⊃ OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(state(q′(s′))))
{-5} transformer_invariant?(pm′‘states,
(memory_read_transformers(pm′‘mem, address_block(addr′, size(uid
∧
transformers_ok?(pm′‘states,
(memory_read_transformers(pm′‘mem, address_block(addr′, size(uidt(dt′)
⊃ OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))
{-6} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′)) ∧
OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(state(q′(s′))))
⊃
data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(state(q′(s′)))) =
data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′))
{-7} interpreted_data_type?[Data](dt′)
{-8} plain_memory?(pm′)
{-9} (address_block(addr′, size(uidt(dt′))) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{-10} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(q′),
address_block(addr′, size(uidt(dt′))))
{-11} pm′‘states(s′)
{-12} OK?(q′(s′))
{-13} CASES memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(s′) OF
OK(sn, bl): dt′‘uidt‘valid?(bl, addr′)
ELSE TRUE
ENDCASES

{1} OK?(CASES memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))(state(q′(s′))) OF
OK(state_1, data): ok_lift(from_byte(dt′)(data, addr′))(state_1),
Exception(ex_type, state_1): Exception(ex_type, state_1),
Fatal: Fatal,
Hang: Hang
ENDCASES)

Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -5,
Using lemma valid_from_byte[Data],
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_read_data_q_ok.1`.

`plain_memory_read_data_q_ok.2:`

```
{-1}  plain_memory?(pm′) ∧
       (address_block(addr′, size(uidt(dt′))) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
       ⊃
       transformer_invariant?(pm′`states,
                              (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪
{-2}  plain_memory?(pm′) ∧
       (address_block(addr′, size(uidt(dt′))) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
       ⊃
       transformers_ok?(pm′`states,
                        (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ memor
{-3}  transformer_invariant?(pm′`states,
                              (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ r
       ∧
       transformers_ok?(pm′`states,
                        (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ memor
        ∧ pm′`states(state(q′(s′)))
       ⊃ OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(state(q′(s′))))
{-4}  transformer_invariant?(pm′`states,
                              (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ r
       ∧
       transformers_ok?(pm′`states,
                        (memory_read_transformers(pm′`mem, address_block(addr′, size(uidt(dt′)))) ∪ memor
       ⊃ OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′))
{-5}  OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′)) ∧
       OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(state(q′(s′))))
       ⊃
       data(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(state(q′(s′)))) =
        data(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′))
{-6}  interpreted_data_type?[Data](dt′)
{-7}  plain_memory?(pm′)
{-8}  (address_block(addr′, size(uidt(dt′))) ⊆ (pm′`ro_addr ∪ pm′`rw_addr))
{-9}  unchanged_memory_invariant?(pm′`mem, pm′`states, singleton(q′),
                                  address_block(addr′, size(uidt(dt′))))
{-10} pm′`states(s′)
{-11} OK?(q′(s′))
{-12} CASES memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(s′) OF
        OK(sn, bl): dt′`uidt`valid?(bl, addr′)
        ELSE TRUE
        ENDCASES
-------------------------------------------------------------------------
 {1}  OK?(memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(state(q′(s′))))
 {2}  OK?(CASES memory_read_list(pm′`mem)(addr′, size(uidt(dt′)))(state(q′(s′))) OF
            OK(state_1, data): ok_lift(from_byte(dt′)(data, addr′))(state_1),
            Exception(ex_type, state_1): Exception(ex_type, state_1),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

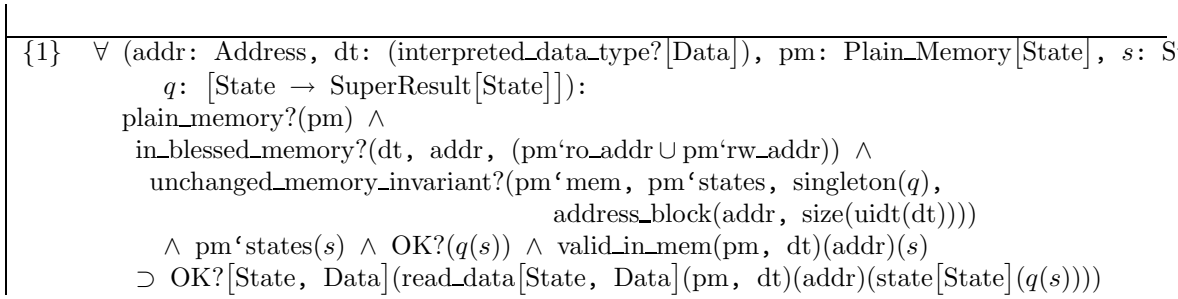
Using lemma unchanged_memory_invariant_invariant[State],
Using lemma super_transformer_invariant_next_ok[State],
Installing automatic rewrites from: (singleton!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_data_q_ok.2`.

Q.E.D.

C.172.14 Plain_Mem_Properties_2.plain_memory_read_data_q_same_TCC1

Terse proof for `plain_memory_read_data_q_same_TCC1`.

`plain_memory_read_data_q_same_TCC1:`

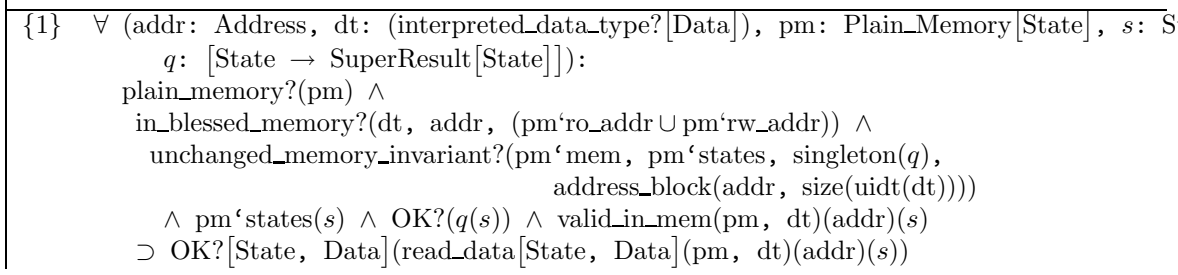


Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_data_q_ok[State, Data],
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_read_data_q_same_TCC1`.
Q.E.D.

C.172.15 Plain_Mem_Properties_2.plain_memory_read_data_q_same_TCC2

Terse proof for `plain_memory_read_data_q_same_TCC2`.

`plain_memory_read_data_q_same_TCC2:`



Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_data_ok,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_read_data_q_same_TCC2`.
Q.E.D.

C.172.16 Plain_Mem_Properties_2.plain_memory_read_data_q_same

Terse proof for `plain_memory_read_data_q_same`.

`plain_memory_read_data_q_same:`

{1} ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
q: [State → SuperResult[State]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt, addr, (pm`ro_addr ∪ pm`rw_addr)) ∧
unchanged_memory_invariant?(pm`mem, pm`states, singleton(q),
address_block(addr, size(uidt(dt))))
∧ pm`states(s) ∧ OK?(q(s)) ∧ valid_in_mem(pm, dt)(addr)(s)
⊃
data(read_data(pm, dt)(addr)(state(q(s)))) =
data(read_data(pm, dt)(addr)(s))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_q_ok,

Simplifying, rewriting, and recording with decision procedures,

Using lemma plain_memory_read_data_ok,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: in_blessed_memory_subset read_data ## expr_2_super expr_2_super_res ok_lift singleton comp_expr_forget_expr

Using lemma super_transformers_ok_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ok_lift,

Expanding the definition of ##,

Using lemma plain_mem_q_read_list[State],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_read_data_q_same`.

Q.E.D.

C.172.17 Plain_Mem_Properties_2.plain_memory_read_write_ok

Terse proof for `plain_memory_read_write_ok`.

`plain_memory_read_write_ok:`

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State],
s: State):
plain_memory?(pm) ∧ pm`states(s) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ⊃
OK?((write_data(pm, dt)(addr, data) ## read_data(pm, dt)(addr))(s))

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_expr_forget_expr ## plain_memory_write_data_ok plain_memory_read_data_ok subset_bigger_union_right

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of ##,

Case splitting on OK?(write_data(pm!1`mem, dt!1)(addr!1, data!1)(s!1)),

we get 2 subgoals:

`plain_memory_read_write_ok.1`:

```
{-1}  OK?(write_data(pm'`mem, dt')(addr', data')(s'))
{-2}  interpreted_data_type?[Data](dt')
{-3}  plain_memory?(pm')
{-4}  pm'`states(s')
{-5}  in_blessed_memory?(dt', addr', pm'`rw_addr)
------------------------------------------------------------------
{1}   OK?(CASES write_data(pm', dt')(addr', data')(s') OF
            OK(state, data): read_data(pm', dt')(addr')(state),
            Exception(ex_type, state): Exception(ex_type, state),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Using lemma plain_memory_read_data_ok,
Using lemma plain_memory_write_data_valid[State, Data],
Expanding the definition of in_blessed_memory?,
Using lemma plain_memory_inv_pred_write_data[State, Data],
Simplifying, rewriting, and recording with decision procedures,
Lifting IF-conditions to the top level,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_read_write_ok.1`.

`plain_memory_read_write_ok.2`:

```
{-1}  interpreted_data_type?[Data](dt')
{-2}  plain_memory?(pm')
{-3}  pm'`states(s')
{-4}  in_blessed_memory?(dt', addr', pm'`rw_addr)
------------------------------------------------------------------
{1}   OK?(write_data(pm'`mem, dt')(addr', data')(s'))
{2}   OK?(CASES write_data(pm', dt')(addr', data')(s') OF
            OK(state, data): read_data(pm', dt')(addr')(state),
            Exception(ex_type, state): Exception(ex_type, state),
            Fatal: Fatal,
            Hang: Hang
            ENDCASES)
```

Hiding formulas: 2,
Using lemma plain_memory_write_data_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of write_data,
Expanding the definition of write_data,
which is trivially true.
This completes the proof of `plain_memory_read_write_ok.2`.
Q.E.D.

C.172.18 Plain_Mem_Properties_2.plain_memory_read_write_res_TCC1

Terse proof for `plain_memory_read_write_res_TCC1`.

`plain_memory_read_write_res_TCC1`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm`states(s) ∧ plain_memory?(pm) ⊃
OK?[State, Data]
((##[State, Unit, Data]
(write_data[State, Data](pm, dt)(addr, data),
read_data[State, Data](pm, dt)(addr)))
(s))

Repeatedly Skolemizing and flattening,

Applying plain_memory_read_write_ok

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `plain_memory_read_write_res_TCC1`.

Q.E.D.

C.172.19 Plain_Mem_Properties_2.plain_memory_read_write_res

Terse proof for `plain_memory_read_write_res`.

`plain_memory_read_write_res`:

{1} ∀ (addr: Address, data1: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧ pm`states(s) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ⊃
data((write_data(pm, dt)(addr, data1) ## read_data(pm, dt)(addr))(s)) = data1

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_write_ok,

Simplifying, rewriting, and recording with decision procedures,

Installing automatic rewrites from: ## write_data read_data ok_lift length_to_byte from_byte_to_byte memory_read_transformers_mono subset_bigger_union_right

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ##,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of ok_lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using plain_mem_write_list_read_list, matching in * where pm gets pm!1,

we get 2 subgoals:

`plain_memory_read_write_res.1:`

{-1} OK?(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data1′, addr′))(s′))
{-2} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data1′, addr′))
(s′))))
{-3} OK?(OK(state(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′,
to_byte(dt′)(data1′, addr′)
(s′)))),
down(from_byte(dt′)(to_byte(dt′)(data1′, addr′), addr′))))
{-4} interpreted_data_type?[Data](dt′)
{-5} plain_memory?(pm′)
{-6} pm′‘states(s′)
{-7} in_blessed_memory?(dt′, addr′, pm′‘rw_addr)

{1} bottom?(from_byte(dt′)(to_byte(dt′)(data1′, addr′), addr′))
{2} down(from_byte(dt′)(to_byte(dt′)(data1′, addr′), addr′)) = data1′

Rewriting using from_byte_to_byte, matching in *,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_read_write_res.1`.

`plain_memory_read_write_res.2:`

{-1} OK?(memory_write_list(pm′‘mem)(addr′, to_byte(dt′)(data1′, addr′))(s′))
{-2} OK?(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′, to_byte(dt′)(data1′, addr′))
(s′))))
{-3} OK?(OK(state(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′,
to_byte(dt′)(data1′, addr′))
(s′)))),
down(from_byte(dt′)
(data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′,
to_byte(dt′)
(data1′, addr′))
(s′)))),
addr′))))
{-4} interpreted_data_type?[Data](dt′)
{-5} plain_memory?(pm′)
{-6} pm′‘states(s′)
{-7} in_blessed_memory?(dt′, addr′, pm′‘rw_addr)

{1} (address_block(addr′, size(uidt(dt′))) ⊆ pm′‘rw_addr)
{2} bottom?(from_byte(dt′)
(data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′,
to_byte(dt′)
(data1′, addr′))
(s′)))),
addr′))
{3} down(from_byte(dt′)
(data(memory_read_list(pm′‘mem)(addr′, size(uidt(dt′)))
(state(memory_write_list(pm′‘mem)
(addr′,
to_byte(dt′)(data1′, addr′))
(s′)))),
addr′))
= data1′

Expanding the definition of in_blessed_memory?,
which is trivially true.
This completes the proof of `plain_memory_read_write_res.2`.
Q.E.D.

C.172.20 Plain_Mem_Properties_2.plain_memory_unchanged_memory_ok_result

Terse proof for `plain_memory_unchanged_memory_ok_result`.

`plain_memory_unchanged_memory_ok_result:`

```
{1}  ∀ (addresses: PRED[Address], pm: Plain_Memory[State], d: Data):
       plain_memory?(pm) ∧ (addresses ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    singleton(expr_2_super[State, Data](ok_result(d))),
                                    addresses)
```

Expanding the definition of unchanged_memory_invariant?,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ok_result!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_unchanged_memory_ok_result.1:`

```
{-1}  plain_memory?(pm′)
{-2}  (addresses′ ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{1}   ∀ (s: State, q: [State → SuperResult[State]], a: Address):
        pm′‘states(s) ∧
         singleton(expr_2_super[State, Data](ok_result(d′)))(q) ∧
          addresses′(a) ∧
           OK?(q(s)) ∧
            OK?(memory_read(pm′‘mem)(a)(s)) ∧
             OK?(memory_read(pm′‘mem)(a)(state(q(s))))
         ⊃
         data(memory_read(pm′‘mem)(a)(state(q(s)))) =
          data(memory_read(pm′‘mem)(a)(s))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -4,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_unchanged_memory_ok_result.1`.

`plain_memory_unchanged_memory_ok_result.2:`

```
{-1}  plain_memory?(pm′)
{-2}  (addresses′ ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{1}   transformer_invariant?(pm′‘states, singleton(expr_2_super[State, Data](ok_result(d′))))
```

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of singleton,

Replacing using formula -4,

Expanding the definition of result_pred,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_unchanged_memory_ok_result.2`.

Q.E.D.

C.173 Proofs for Plain_Mem_Properties_3 (plain_memory.pvs)

C.173.1 Plain_Mem_Properties_3.plain_memory_read_write_other_ok

Terse proof for `plain_memory_read_write_other_ok`.

`plain_memory_read_write_other_ok`:

{1} ∀ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): plain_memory?(pm) ∧ pm‘states(s) ∧ in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧ in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ valid_in_mem(pm, dt2)(addr2)(s) ⊃ OK?((write_data(pm, dt1)(addr1, data1) ## read_data(pm, dt2)(addr2))(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of ##,
Case splitting on OK?(write_data(pm!1, dt1!1)(addr1!1, data1!1)(s!1)),
we get 2 subgoals:

`plain_memory_read_write_other_ok.1`:

{-1} OK?(write_data(pm′, dt1′)(addr1′, data1′)(s′))
{-2} interpreted_data_type?[Data1](dt1′)
{-3} interpreted_data_type?[Data2](dt2′)
{-4} plain_memory?(pm′)
{-5} pm′‘states(s′)
{-6} in_blessed_memory?(dt1′, addr1′, pm′‘rw_addr)
{-7} in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8} blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-9} valid_in_mem(pm′, dt2′)(addr2′)(s′)

{1} OK?(CASES write_data(pm′, dt1′)(addr1′, data1′)(s′) OF
OK(state, data): read_data(pm′, dt2′)(addr2′)(state),
Exception(ex_type, state): Exception(ex_type, state),
Fatal: Fatal,
Hang: Hang
ENDCASES)

Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_memory_read_data_q_ok[State, Data2],
Installing automatic rewrites from: (expr_2_super! expr_2_super_res!)
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_unchanged_memory_invariant_write_data[State, Data1],
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_reflexive, matching in *,
Expanding the definition of in_blessed_memory?,
Keeping (-3 -9 -10 1) and hiding *,
Using lemma blocks_in_larger_set,

Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_bigger_union_right, matching in *,
This completes the proof of `plain_memory_read_write_other_ok.1`.
`plain_memory_read_write_other_ok.2:`

```
{-1}  interpreted_data_type?[Data1](dt1′)
{-2}  interpreted_data_type?[Data2](dt2′)
{-3}  plain_memory?(pm′)
{-4}  pm′‘states(s′)
{-5}  in_blessed_memory?(dt1′, addr1′, pm′‘rw_addr)
{-6}  in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7}  blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-8}  valid_in_mem(pm′, dt2′)(addr2′)(s′)
------------------------------------------------------------------------
{1}   OK?(write_data(pm′, dt1′)(addr1′, data1′)(s′))
{2}   OK?(CASES write_data(pm′, dt1′)(addr1′, data1′)(s′) OF
            OK(state, data): read_data(pm′, dt2′)(addr2′)(state),
            Exception(ex_type, state): Exception(ex_type, state),
            Fatal: Fatal,
            Hang: Hang
          ENDCASES)
```

Rewriting using plain_memory_write_data_ok, matching in *,
This completes the proof of `plain_memory_read_write_other_ok.2`.
Q.E.D.

C.173.2 Plain_Mem_Properties_3.plain_memory_read_write_other_res_TCC1

Terse proof for `plain_memory_read_write_other_res_TCC1`.
`plain_memory_read_write_other_res_TCC1:`

```
------------------------------------------------------------------------
{1}   ∀ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]),
           dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
        valid_in_mem(pm, dt2)(addr2)(s) ∧
         blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
          in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
           in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧ pm‘states(s) ∧ plain_memory?(pm)
        ⊃
        OK?[State, Data2]
            ((##[State, Unit, Data2]
                 (write_data[State, Data1](pm, dt1)(addr1, data1),
                   read_data[State, Data2](pm, dt2)(addr2)))
               (s))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_write_other_ok,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_read_write_other_res_TCC1`.
Q.E.D.

C.173.3 Plain_Mem_Properties_3.plain_memory_read_write_other_res_TCC2

Terse proof for `plain_memory_read_write_other_res_TCC2`.

`plain_memory_read_write_other_res_TCC2`:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
valid_in_mem(pm, dt2)(addr2)(s) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧ pm‘states(s) ∧ plain_memory?(pm)
⊃ OK?[State, Data2](read_data[State, Data2](pm, dt2)(addr2)(s))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok[State, Data2],

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_read_write_other_res_TCC2`.

Q.E.D.

C.173.4 Plain_Mem_Properties_3.plain_memory_read_write_other_res

Terse proof for `plain_memory_read_write_other_res`.

`plain_memory_read_write_other_res`:

{1} ∀ (addr1, addr2: Address, data1: Data1, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧
pm‘states(s) ∧
in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
valid_in_mem(pm, dt2)(addr2)(s)
⊃
data((write_data(pm, dt1)(addr1, data1) ## read_data(pm, dt2)(addr2))(s)) =
data(read_data(pm, dt2)(addr2)(s))

Installing automatic rewrites from: (## expr_2_super expr_2_super_res)

Repeatedly Skolemizing and flattening,

Case splitting on OK?(write_data(pm!1, dt1!1)(addr1!1, data1!1)(s!1)),

we get 2 subgoals:

`plain_memory_read_write_other_res.1:`

{-1}	OK?(write_data(pm′, dt1′)(addr1′, data1′)(s′))
{-2}	interpreted_data_type?[Data1](dt1′)
{-3}	interpreted_data_type?[Data2](dt2′)
{-4}	plain_memory?(pm′)
{-5}	pm′‘states(s′)
{-6}	in_blessed_memory?(dt1′, addr1′, pm′‘rw_addr)
{-7}	in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8}	blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-9}	valid_in_mem(pm′, dt2′)(addr2′)(s′)
{1}	data((write_data(pm′, dt1′)(addr1′, data1′) ## read_data(pm′, dt2′)(addr2′))(s′)) = data(read_data(pm′, dt2′)(addr2′)(s′))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_read_data_q_same[State, Data2],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_unchanged_memory_invariant_write_data[State, Data1],
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_reflexive, matching in *,
Expanding the definition of in_blessed_memory?,
Keeping (-3 -9 -10 1) and hiding *,
Using lemma blocks_in_larger_set,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_bigger_union_right, matching in *,
This completes the proof of `plain_memory_read_write_other_res.1`.
`plain_memory_read_write_other_res.2:`

{-1}	interpreted_data_type?[Data1](dt1′)
{-2}	interpreted_data_type?[Data2](dt2′)
{-3}	plain_memory?(pm′)
{-4}	pm′‘states(s′)
{-5}	in_blessed_memory?(dt1′, addr1′, pm′‘rw_addr)
{-6}	in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7}	blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-8}	valid_in_mem(pm′, dt2′)(addr2′)(s′)
{1}	OK?(write_data(pm′, dt1′)(addr1′, data1′)(s′))
{2}	data((write_data(pm′, dt1′)(addr1′, data1′) ## read_data(pm′, dt2′)(addr2′))(s′)) = data(read_data(pm′, dt2′)(addr2′)(s′))

Rewriting using plain_memory_write_data_ok[State, Data1], matching in *,
This completes the proof of `plain_memory_read_write_other_res.2`.
Q.E.D.

C.173.5 Plain_Mem_Properties_3.plain_memory_read_read_ok

Terse proof for `plain_memory_read_read_ok`.

`plain_memory_read_read_ok:`

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧
pm‘states(s) ∧
in_blessed_memory?(dt1, addr1, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
valid_in_mem(pm, dt1)(addr1)(s) ∧ valid_in_mem(pm, dt2)(addr2)(s)
⊃ OK?((read_data(pm, dt1)(addr1) ## read_data(pm, dt2)(addr2))(s))

Repeatedly Skolemizing and flattening,

Case splitting on OK?(read_data(pm!1, dt1!1)(addr1!1)(s!1)),

we get 2 subgoals:

`plain_memory_read_read_ok.1:`

{-1} OK?(read_data(pm′, dt1′)(addr1′)(s′))
{-2} interpreted_data_type?[Data1](dt1′)
{-3} interpreted_data_type?[Data2](dt2′)
{-4} plain_memory?(pm′)
{-5} pm′‘states(s′)
{-6} in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-7} in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-8} valid_in_mem(pm′, dt1′)(addr1′)(s′)
{-9} valid_in_mem(pm′, dt2′)(addr2′)(s′)

{1} OK?((read_data(pm′, dt1′)(addr1′) ## read_data(pm′, dt2′)(addr2′))(s′))

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_read_data_q_ok[State, Data2],

Instantiating the top quantifier in -1 with the terms: expr_2_super(read_data(pm′, dt1′)(addr1′)),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_memory_invariant_read_data[State, Data1],

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_read_read_ok.1.1:`

{-1}	unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), (pm′‘ro_addr ∪ pm′‘rw_addr))
{-2}	plain_memory?(pm′)
{-3}	in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4}	in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}	pm′‘states(s')
{-6}	valid_in_mem(pm′, dt2′)(addr2′)(s')
{-7}	OK?(read_data(pm′, dt1′)(addr1′)(s'))
{-8}	interpreted_data_type?⌈Data1⌉(dt1′)
{-9}	interpreted_data_type?⌈Data2⌉(dt2′)
{-10}	valid_in_mem(pm′, dt1′)(addr1′)(s')
{1}	(address_block(addr2′, size(uidt(dt2′))) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{2}	unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), address_block(addr2′, size(uidt(dt2′))))
{3}	OK?(read_data(pm′, dt2′)(addr2′)(state(read_data(pm′, dt1′)(addr1′)(s'))))

Expanding the definition of in_blessed_memory?,

which is trivially true.

This completes the proof of `plain_memory_read_read_ok.1.1`.

`plain_memory_read_read_ok.1.2:`

{-1}	unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), (pm′‘ro_addr ∪ pm′‘rw_addr))
{-2}	plain_memory?(pm′)
{-3}	in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4}	in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}	pm′‘states(s')
{-6}	valid_in_mem(pm′, dt2′)(addr2′)(s')
{-7}	OK?(read_data(pm′, dt1′)(addr1′)(s'))
{-8}	interpreted_data_type?⌈Data1⌉(dt1′)
{-9}	interpreted_data_type?⌈Data2⌉(dt2′)
{-10}	valid_in_mem(pm′, dt1′)(addr1′)(s')
{1}	(singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))) ⊆ singleton(expr_2_super(read_data(pm′,
{2}	unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), address_block(addr2′, size(uidt(dt2′))))
{3}	OK?(read_data(pm′, dt2′)(addr2′)(state(read_data(pm′, dt1′)(addr1′)(s'))))

Rewriting using subset_reflexive, matching in *,

This completes the proof of `plain_memory_read_read_ok.1.2`.

`plain_memory_read_read_ok.2:`

{-1} interpreted_data_type?[Data1](dt1′)
{-2} interpreted_data_type?[Data2](dt2′)
{-3} plain_memory?(pm′)
{-4} pm′`states(s′)
{-5} in_blessed_memory?(dt1′, addr1′, (pm′`ro_addr ∪ pm′`rw_addr))
{-6} in_blessed_memory?(dt2′, addr2′, (pm′`ro_addr ∪ pm′`rw_addr))
{-7} valid_in_mem(pm′, dt1′)(addr1′)(s′)
{-8} valid_in_mem(pm′, dt2′)(addr2′)(s′)

{1} OK?(read_data(pm′, dt1′)(addr1′)(s′))
{2} OK?((read_data(pm′, dt1′)(addr1′) ## read_data(pm′, dt2′)(addr2′))(s′))

Rewriting using plain_memory_read_data_ok[State, Data1], matching in *,

This completes the proof of `plain_memory_read_read_ok.2`.

Q.E.D.

C.173.6 Plain_Mem_Properties_3.plain_memory_read_read_TCC1

Terse proof for `plain_memory_read_read_TCC1`.

`plain_memory_read_read_TCC1:`

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
valid_in_mem(pm, dt2)(addr2)(s) ∧
valid_in_mem(pm, dt1)(addr1)(s) ∧
in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
pm`states(s) ∧ plain_memory?(pm)
⊃
OK?[State, Data2]
((##[State, Data1, Data2]
(read_data[State, Data1](pm, dt1)(addr1),
read_data[State, Data2](pm, dt2)(addr2)))
(s))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_read_ok,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_read_read_TCC1`.

Q.E.D.

C.173.7 Plain_Mem_Properties_3.plain_memory_read_read_TCC2

Terse proof for `plain_memory_read_read_TCC2`.

`plain_memory_read_read_TCC2:`

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       valid_in_mem(pm, dt2)(addr2)(s) ∧
        valid_in_mem(pm, dt1)(addr1)(s) ∧
         in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
          in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
           pm`states(s) ∧ plain_memory?(pm)
        ⊃ OK?[State, Data2](read_data[State, Data2](pm, dt2)(addr2)(s))
```

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok[State, Data2],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_read_read_TCC2`.

Q.E.D.

C.173.8 Plain_Mem_Properties_3.plain_memory_read_read

Terse proof for `plain_memory_read_read`.

`plain_memory_read_read:`

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        pm`states(s) ∧
         in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
          in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
           valid_in_mem(pm, dt1)(addr1)(s) ∧ valid_in_mem(pm, dt2)(addr2)(s)
        ⊃
        data((read_data(pm, dt1)(addr1) ## read_data(pm, dt2)(addr2))(s)) =
         data(read_data(pm, dt2)(addr2)(s))
```

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_read_data_ok[State, Data1],

Installing automatic rewrites from: (expr_2_super! expr_2_super_res! ##!)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_read_data_q_same,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Using lemma plain_memory_unchanged_memory_invariant_read_data[State, Data1],

Using lemma unchanged_memory_invariant_mono,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_read_read.1`:

{-1} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), (pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} plain_memory?(pm′)
{-3} in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5} pm′‘states(s′)
{-6} valid_in_mem(pm′, dt2′)(addr2′)(s′)
{-7} valid_in_mem(pm′, dt1′)(addr1′)(s′)
{-8} OK?(read_data(pm′, dt1′)(addr1′)(s′))

{1} (address_block(addr2′, size(uidt(dt2′))) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
{2} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), address_block(addr2′, size(uidt(dt2′))))
{3} data(read_data(pm′, dt2′)(addr2′)(state(read_data(pm′, dt1′)(addr1′)(s′))))
= data(read_data(pm′, dt2′)(addr2′)(s′))

Expanding the definition of in_blessed_memory?,

which is trivially true.

This completes the proof of `plain_memory_read_read.1`.

`plain_memory_read_read.2`:

{-1} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), (pm′‘ro_addr ∪ pm′‘rw_addr))
{-2} plain_memory?(pm′)
{-3} in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5} pm′‘states(s′)
{-6} valid_in_mem(pm′, dt2′)(addr2′)(s′)
{-7} valid_in_mem(pm′, dt1′)(addr1′)(s′)
{-8} OK?(read_data(pm′, dt1′)(addr1′)(s′))

{1} (singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))) ⊆ singleton(expr_2_super(read_data(pm′, dt1′)(addr1′
{2} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(read_data(pm′, dt1′)(addr1′))), address_block(addr2′, size(uidt(dt2′))))
{3} data(read_data(pm′, dt2′)(addr2′)(state(read_data(pm′, dt1′)(addr1′)(s′))))
= data(read_data(pm′, dt2′)(addr2′)(s′))

Rewriting using subset_reflexive, matching in *,

This completes the proof of `plain_memory_read_read.2`.

Q.E.D.

C.173.9 Plain_Mem_Properties_3.plain_memory_unchanged_composition

Terse proof for `plain_memory_unchanged_composition`.

`plain_memory_unchanged_composition:`

```
{1}  ∀ (addresses: PRED[Address], pm: Plain_Memory[State],
          q: [State → ExprResult[State, Data1]],
          r: [Data1 → [State → ExprResult[State, Data2]]], P: PRED[Data1]):
       (plain_memory?(pm) ∧
          (addresses ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ∧
           (∀ (s: (pm‘states)): OK?(q(s)) ⊃ P(data(q(s)))) ∧
            unchanged_memory_invariant?(pm‘mem, pm‘states, single-
     ton(expr_2_super(q)), addresses)
              ∧
              (∀ (d: (P)):
                   unchanged_memory_invariant?(pm‘mem, pm‘states, single-
     ton(expr_2_super(r(d))),
                                                          addresses)))
       ⊃
       unchanged_memory_invariant?(pm‘mem, pm‘states, single-
     ton(expr_2_super(q ## r)),
                                          addresses)
```

Repeatedly Skolemizing and flattening,

Using lemma fexpr_unchanged_memory_invariant_composition,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`plain_memory_unchanged_composition.1:`

```
{-1}  unchanged_memory_invariant?(pm′‘mem, pm′‘states, single-
      ton(expr_2_super(q′)), addresses′)
{-2}  ∀ (s: (pm′‘states)): OK?(q′(s)) ⊃ P′(data(q′(s)))
{-3}  ∀ (d: (P′)):
        unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(r′(d))),
                                    addresses′)
{-4}  plain_memory?(pm′)
{-5}  (addresses′ ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))
-------------------------------------------------------------------------------
{1}   transformers_ok?(pm′‘states, memory_read_transformers(pm′‘mem, addresses′))
{2}   unchanged_memory_invariant?(pm′‘mem, pm′‘states, single-
      ton(expr_2_super(q′ ## r′)),
                                    addresses′)
```

Keeping (-4 -5 1) and hiding *,

Using lemma plain_memory_transformers_ok_read_ro_rw,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using transformers_ok_mono_transformers, matching in * where states gets pm!1‘states, transformers_1 gets memory_read_transformers(pm!1‘mem, addresses!1), transformers_2 gets memory_read_transformers(pm!1‘mem, union(pm!1‘ro_addr, pm!1‘rw_addr)),

Rewriting using memory_read_transformers_mono, matching in *,

This completes the proof of `plain_memory_unchanged_composition.1`.

`plain_memory_unchanged_composition.2:`

{-1} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(q')), addresses′)
{-2} $\forall$ (s: (pm′‘states)): OK?($q'(s)$) $\supset$ P'(data($q'(s)$))
{-3} $\forall$ (d: (P')): unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super($r'(d)$)), addresses′)
{-4} plain_memory?(pm′)
{-5} (addresses′ $\subseteq$ (pm′‘ro_addr $\cup$ pm′‘rw_addr))

{1} singleton(expr_2_super(q'))(expr_2_super(q'))
{2} unchanged_memory_invariant?(pm′‘mem, pm′‘states, singleton(expr_2_super(q' ## r')), addresses′)

Expanding the definition of singleton,
which is trivially true.
This completes the proof of `plain_memory_unchanged_composition.2`.
Q.E.D.

C.174 Proofs for Plain_Mem_Rewrites (plain_memory_rewrites.pvs)

This theory contains no provable formal statements.

C.175 Proofs for Plain_Mem_Rewrites1 (plain_memory_rewrites.pvs)

C.175.1 Plain_Mem_Rewrites1.pm_q_prop_TCC1

Terse proof for `pm_q_prop_TCC1`.

`pm_q_prop_TCC1:`

{1} $\forall$ (res: StmtResult[State]): OK?(res) $\supset$ OK?[State](res) $\vee$ Break?[State](res) $\vee$ Continue?[State](res) $\vee$ Return?[State](res) $\vee$ Switch?[State](res) $\vee$ Default?[State](res) $\vee$ Exception?[State](res)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_TCC1`.
Q.E.D.

C.175.2 Plain_Mem_Rewrites1.pm_q_prop_ok_stmt

Terse proof for `pm_q_prop_ok_stmt`.

pm_q_prop_ok_stmt:

{1} ∀ (pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]):
plain_memory?(pm) ∧ pm_q_prop(pm)(q(s)) ⊃ OK?(q(s))

Expanding the definition of pm_q_prop,
Repeatedly Skolemizing and flattening,
This completes the proof of pm_q_prop_ok_stmt.
Q.E.D.

C.176 Proofs for Plain_Mem_Rewrites2 (plain_memory_rewrites.pvs)

C.176.1 Plain_Mem_Rewrites2.in_blessed_memory_rw_ro

Terse proof for in_blessed_memory_rw_ro.
in_blessed_memory_rw_ro:

{1} ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]):
in_blessed_memory?(dt, addr, pm‘rw_addr) ⊃
in_blessed_memory?(dt, addr, (pm‘ro_addr ∪ pm‘rw_addr))

Repeatedly Skolemizing and flattening,
Expanding the definition of in_blessed_memory?,
Rewriting using subset_bigger_union_right, matching in *,
This completes the proof of in_blessed_memory_rw_ro.
Q.E.D.

C.176.2 Plain_Mem_Rewrites2.pm_q_prop_TCC1

Terse proof for pm_q_prop_TCC1.
pm_q_prop_TCC1:

{1} ∀ (res: ExprResult[State, Data]):
OK?(res) ⊃ OK?[State, Data](res) ∨ Exception?[State, Data](res)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_q_prop_TCC1.
Q.E.D.

C.176.3 Plain_Mem_Rewrites2.pm_q_prop_ok_expr

Terse proof for pm_q_prop_ok_expr.
pm_q_prop_ok_expr:

{1} ∀ (pm: Plain_Memory[State], s: State, q: [State → ExprResult[State, Data]]):
plain_memory?(pm) ∧ pm_q_prop(pm)(q(s)) ⊃ OK?(q(s))

Expanding the definition of pm_q_prop,
Repeatedly Skolemizing and flattening,

This completes the proof of `pm_q_prop_ok_expr`.
Q.E.D.

C.176.4 Plain_Mem_Rewrites2.pm_q_prop_single_read

Terse proof for `pm_q_prop_single_read`.
`pm_q_prop_single_read`:

{1} ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧
pm‘states(s) ∧
in_blessed_memory?(dt, addr, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
valid_in_mem(pm, dt)(addr)(s)
⊃ pm_q_prop(pm)(read_data(pm, dt)(addr)(s))

Expanding the definition of pm_q_prop,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformer_invariant_read_data[State, Data],
Using lemma expr_transformer_invariant_next_ok[State, Data],
Using lemma plain_memory_read_data_ok[State, Data],
Expanding the definition of has_next_state,
Expanding the definition of singleton,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_q_prop_single_read`.
Q.E.D.

C.176.5 Plain_Mem_Rewrites2.pm_q_prop_ok_result_single

Terse proof for `pm_q_prop_ok_result_single`.
`pm_q_prop_ok_result_single`:

{1} ∀ (data: Data, pm: Plain_Memory[State], s: State):
pm‘states(s) ⊃ pm_q_prop(pm)(ok_result(data)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_ok_result_single`.
Q.E.D.

C.176.6 Plain_Mem_Rewrites2.plain_memory_write_ok_single

Terse proof for `plain_memory_write_ok_single`.
`plain_memory_write_ok_single`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State],
s: State):
plain_memory?(pm) ∧ pm‘states(s) ∧ in_blessed_memory?(dt, addr, pm‘rw_addr) ⊃
OK?(write_data(pm, dt)(addr, data)(s))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_write_data_ok, matching in *,
This completes the proof of `plain_memory_write_ok_single`.
Q.E.D.

C.176.7 Plain_Mem_Rewrites2.pm_q_prop_read_TCC1

Terse proof for `pm_q_prop_read_TCC1`.

`pm_q_prop_read_TCC1:`

```
{1}  ∀ (dt: (interpreted_data_type?[Data]), res: StmtResult[State]):
       OK?(res) ⊃
        OK?[State](res) ∨
         Break?[State](res) ∨
          Continue?[State](res) ∨
           Return?[State](res) ∨
            Switch?[State](res) ∨ Default?[State](res) ∨ Exception?[State](res)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_read_TCC1`.
Q.E.D.

C.177 Proofs for Plain_Mem_Rewrites3 (plain_memory_rewrites.pvs)

C.177.1 Plain_Mem_Rewrites3.pm_q_prop_read_TCC1

Terse proof for `pm_q_prop_read_TCC1`.

`pm_q_prop_read_TCC1:`

```
{1}  ∀ (dt: (interpreted_data_type?[Data]), res: ExprResult[State, Data_q]):
       OK?(res) ⊃ OK?[State, Data_q](res) ∨ Exception?[State, Data_q](res)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_read_TCC1`.
Q.E.D.

C.178 Proofs for Plain_Mem_Rewrites4 (plain_memory_rewrites.pvs)

C.178.1 Plain_Mem_Rewrites4.plain_memory_write_ok_q_expr

Terse proof for `plain_memory_write_ok_q_expr`.

`plain_memory_write_ok_q_expr:`

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]
         s: State, q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm_q_prop(pm)(q(s))
        ⊃ OK?((q ## write_data(pm, dt)(addr, data))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of ##,

Expanding the definition of pm_q_prop,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using plain_memory_write_data_ok, matching in *,
This completes the proof of `plain_memory_write_ok_q_expr`.
Q.E.D.

C.178.2 Plain_Mem_Rewrites4.pm_q_prop_read_ok_expr

Terse proof for `pm_q_prop_read_ok_expr`.

`pm_q_prop_read_ok_expr`:

```
{1}  ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧ pm_q_prop_read(pm, dt, addr)(q(s)) ⊃
        OK?((q ## read_data(pm, dt)(addr))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Expanding the definition of ##,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_q_prop_read_ok_expr`.
Q.E.D.

C.178.3 Plain_Mem_Rewrites4.pm_q_prop_read_pm_q_prop_expr

Terse proof for `pm_q_prop_read_pm_q_prop_expr`.

`pm_q_prop_read_pm_q_prop_expr`:

```
{1}  ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt, addr, (pm`ro_addr ∪ pm`rw_addr)) ∧
         pm_q_prop_read(pm, dt, addr)(q(s))
        ⊃ pm_q_prop(pm)((q ## read_data(pm, dt)(addr))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of ##,
Expanding the definition of pm_q_prop,
Expanding the definition of pm_q_prop_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_transformer_invariant_read_data[State, Data],
Using lemma expr_transformer_invariant_next_ok[State, Data],
Expanding the definition of has_next_state,
Expanding the definition of singleton,
which is trivially true.
This completes the proof of `pm_q_prop_read_pm_q_prop_expr`.
Q.E.D.

C.178.4 Plain_Mem_Rewrites4.pm_q_prop_read_write_q_expr

Terse proof for `pm_q_prop_read_write_q_expr`.

`pm_q_prop_read_write_q_expr:`

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]
          s: State, q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm‘rw_addr) ∧ pm_q_prop(pm)(q(s))
        ⊃ pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of ##,
Expanding the definition of pm_q_prop,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pm_q_prop_read,
Using lemma plain_memory_write_data_ok[State, Data],
Using lemma plain_memory_transformer_invariant_write_data[State, Data],
Using lemma expr_transformer_invariant_next_ok,
Expanding the definition of singleton,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_read_write_ok[State, Data],
Expanding the definition of ##,
Expanding the definition of ##,
which is trivially true.
This completes the proof of `pm_q_prop_read_write_q_expr`.
Q.E.D.

C.178.5 Plain_Mem_Rewrites4.pm_q_prop_ok_result_q_expr

Terse proof for `pm_q_prop_ok_result_q_expr`.

`pm_q_prop_ok_result_q_expr:`

```
{1}  ∀ (data: Data, pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧ pm_q_prop(pm)(q(s)) ⊃
        pm_q_prop(pm)((q ## ok_result[State, Data](data))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_ok_result_q_expr`.
Q.E.D.

C.178.6 Plain_Mem_Rewrites4.pm_q_prop_read_ok_result_single

Terse proof for `pm_q_prop_read_ok_result_single`.

pm_q_prop_read_ok_result_single:

```
{1}  ∀ (addr: Address, data_q: Data_q, dt: (interpreted_data_type?[Data]),
          pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        pm`states(s) ∧
         in_blessed_memory?(dt, addr, (pm`ro_addr ∪ pm`rw_addr)) ∧
          valid_in_mem(pm, dt)(addr)(s)
        ⊃ pm_q_prop_read(pm, dt, addr)(ok_result[State, Data_q](data_q)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma pm_q_prop_ok_result_single[State, Data_q],
Expanding the definition of pm_q_prop,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using plain_memory_read_data_ok, matching in *,
Rewriting using ok_result_state, matching in *,
This completes the proof of pm_q_prop_read_ok_result_single.
Q.E.D.

C.178.7 Plain_Mem_Rewrites4.plain_memory_read_write_q_data_expr_TCC1

Terse proof for plain_memory_read_write_q_data_expr_TCC1.

plain_memory_read_write_q_data_expr_TCC1:

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State],
          s: State, q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm_q_prop(pm)(q(s))
        ⊃
        OK?[State, Data]
            ((##[State, Unit, Data]
                 (##[State, Data_q, Unit]
                      (q, write_data[State, Data](pm, dt)(addr, data)),
                    read_data[State, Data](pm, dt)(addr)))
               (s))
```

Repeatedly Skolemizing and flattening,
Using lemma pm_q_prop_read_write_q_expr,
Expanding the definition of pm_q_prop_read,
Installing automatic rewrites from: ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_read_write_q_data_expr_TCC1.
Q.E.D.

C.178.8 Plain_Mem_Rewrites4.plain_memory_read_write_q_data_expr

Terse proof for plain_memory_read_write_q_data_expr.

plain_memory_read_write_q_data_expr:

```
{1}  ∀ (addr: Address, data1: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State
         s: State, q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm‘rw_addr) ∧ pm_q_prop(pm)(q(s))
        ⊃
        data((q ## write_data(pm, dt)(addr, data1) ## read_data(pm, dt)(addr))(s)) =
         data1
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: ((##!!))
Expanding the definition of pm_q_prop,
Using lemma plain_memory_read_write_res[State, Data],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_read_write_q_data_expr.
Q.E.D.

C.178.9 Plain_Mem_Rewrites4.pm_q_prop_read_write_other_single

Terse proof for pm_q_prop_read_write_other_single.

pm_q_prop_read_write_other_single:

```
{1}  ∀ (addr1, addr2: Address, data: Data, dt1: (interpreted_data_type?[Data]),
          dt2: (interpreted_data_type?[Data_q]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
         in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
          in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
           valid_in_mem(pm, dt2)(addr2)(s) ∧ pm‘states(s)
        ⊃ pm_q_prop_read(pm, dt2, addr2)(write_data(pm, dt1)(addr1, data)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma plain_memory_write_data_ok[State, Data],
Using lemma plain_memory_read_write_other_ok[State, Data, Data_q],
Installing automatic rewrites from: ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_transformer_invariant_write_data[State, Data],
Using lemma expr_transformer_invariant_next_ok,
Expanding the definition of singleton,
Expanding the definition of has_next_state,
which is trivially true.
This completes the proof of pm_q_prop_read_write_other_single.
Q.E.D.

C.178.10 Plain_Mem_Rewrites4.pm_q_prop_read_read_single

Terse proof for pm_q_prop_read_read_single.

pm_q_prop_read_read_single:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data]),
dt2: (interpreted_data_type?[Data_q]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
valid_in_mem(pm, dt1)(addr1)(s) ∧
valid_in_mem(pm, dt2)(addr2)(s) ∧ pm`states(s)
⊃ pm_q_prop_read(pm, dt2, addr2)(read_data(pm, dt1)(addr1)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma pm_q_prop_single_read[State, Data],
Expanding the definition of pm_q_prop,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_read_read_ok[State, Data, Data_q],
Installing automatic rewrites from: ##
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pm_q_prop_read_read_single.
Q.E.D.

C.179 Proofs for Plain_Mem_Rewrites5 (plain_memory_rewrites.pvs)

C.179.1 Plain_Mem_Rewrites5.plain_memory_write_ok_q_stmt

Terse proof for plain_memory_write_ok_q_stmt.

plain_memory_write_ok_q_stmt:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State],
s: State, q: [State → StmtResult[State]]):
plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm_q_prop(pm)(q(s))
⊃ OK?((q ## write_data(pm, dt)(addr, data))(s))

Installing automatic rewrites from: ##
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pm_q_prop,
Expanding the definition of lift,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Rewriting using plain_memory_write_data_ok, matching in *,
This completes the proof of plain_memory_write_ok_q_stmt.
Q.E.D.

C.179.2 Plain_Mem_Rewrites5.pm_q_prop_read_ok_stmt

Terse proof for pm_q_prop_read_ok_stmt.

pm_q_prop_read_ok_stmt:

```
{1}  ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
          q: [State → StmtResult[State]]):
       plain_memory?(pm) ∧ pm_q_prop_read(pm, dt, addr)(q(s)) ⊃
        OK?((q ## read_data(pm, dt)(addr))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of lift,
Expanding the definition of pm_q_prop_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pm_q_prop_read_ok_stmt.
Q.E.D.

C.179.3 Plain_Mem_Rewrites5.pm_q_prop_read_pm_q_prop_stmt

Terse proof for pm_q_prop_read_pm_q_prop_stmt.

pm_q_prop_read_pm_q_prop_stmt:

```
{1}  ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State,
          q: [State → StmtResult[State]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt, addr, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
         pm_q_prop_read(pm, dt, addr)(q(s))
        ⊃ pm_q_prop(pm)((q ## read_data(pm, dt)(addr))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of pm_q_prop,
Expanding the definition of pm_q_prop_read,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

pm_q_prop_read_pm_q_prop_stmt.1:

```
{-1}  interpreted_data_type?[Data](dt′)
{-2}  plain_memory?(pm′)
{-3}  in_blessed_memory?(dt′, addr′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4}  OK?(q′(s′))
{-5}  OK?(read_data(pm′, dt′)(addr′)(state(q′(s′))))
{-6}  pm′‘states(state(q′(s′)))
-------------------------------------------------------------------
 {1}  pm′‘states(state(lift(read_data(pm′, dt′)(addr′))(q′(s′))))
```

Using lemma plain_memory_transformer_invariant_read_data[State, Data],
Using lemma expr_transformer_invariant_next_ok[State, Data],
Expanding the definition of has_next_state,
Expanding the definition of singleton,
Expanding the definition of lift,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pm_q_prop_read_pm_q_prop_stmt.1.

pm_q_prop_read_pm_q_prop_stmt.2:

{-1} interpreted_data_type?[Data](dt′)
{-2} plain_memory?(pm′)
{-3} in_blessed_memory?(dt′, addr′, (pm′`ro_addr ∪ pm′`rw_addr))
{-4} OK?($q'(s')$)
{-5} OK?(read_data(pm′, dt′)(addr′)(state($q'(s')$)))
{-6} pm′`states(state($q'(s')$))

{1} OK?(lift(read_data(pm′, dt′)(addr′))($q'(s')$))

Expanding the definition of lift,
which is trivially true.
This completes the proof of pm_q_prop_read_pm_q_prop_stmt.2.
Q.E.D.

C.179.4 Plain_Mem_Rewrites5.pm_q_prop_read_write_q_stmt

Terse proof for pm_q_prop_read_write_q_stmt.
pm_q_prop_read_write_q_stmt:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]):
plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm_q_prop(pm)($q(s)$)
⊃ pm_q_prop_read(pm, dt, addr)((q ## write_data(pm, dt)(addr, data))(s))

Installing automatic rewrites from: ##
Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of lift,
Expanding the definition of pm_q_prop,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pm_q_prop_read,
Using lemma plain_memory_write_data_ok[State, Data],
Using lemma plain_memory_transformer_invariant_write_data[State, Data],
Using lemma expr_transformer_invariant_next_ok,
Expanding the definition of singleton,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma plain_memory_read_write_ok[State, Data],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of pm_q_prop_read_write_q_stmt.
Q.E.D.

C.179.5 Plain_Mem_Rewrites5.pm_q_prop_ok_result_q_stmt

Terse proof for pm_q_prop_ok_result_q_stmt.
pm_q_prop_ok_result_q_stmt:

{1} ∀ (data: Data, pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]):
plain_memory?(pm) ∧ pm_q_prop(pm)($q(s)$) ⊃
pm_q_prop(pm)((q ## ok_result[State, Data](data))(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_ok_result_q_stmt`.
Q.E.D.

C.179.6 Plain_Mem_Rewrites5.plain_memory_read_write_q_data_stmt_TCC1

Terse proof for `plain_memory_read_write_q_data_stmt_TCC1`.

`plain_memory_read_write_q_data_stmt_TCC1:`

```
{1}  ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State]
         s: State, q: [State → StmtResult[State]]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm_q_prop(pm)(q(s))
        ⊃
        OK?[State, Data]
            ((##[State, Unit, Data]
                  (##[State, Unit](q, write_data[State, Data](pm, dt)(addr, data)),
                    read_data[State, Data](pm, dt)(addr)))
                (s))
```

Repeatedly Skolemizing and flattening,
Using lemma pm_q_prop_read_write_q_stmt,
Expanding the definition of pm_q_prop_read,
Installing automatic rewrites from: ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_write_q_data_stmt_TCC1`.
Q.E.D.

C.179.7 Plain_Mem_Rewrites5.plain_memory_read_write_q_data_stmt

Terse proof for `plain_memory_read_write_q_data_stmt`.

`plain_memory_read_write_q_data_stmt:`

```
{1}  ∀ (addr: Address, data1: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State
         s: State, q: [State → StmtResult[State]]):
       plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm_q_prop(pm)(q(s))
        ⊃
        data((q ## write_data(pm, dt)(addr, data1) ## read_data(pm, dt)(addr))(s)) =
         data1
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: ((##!!))
Expanding the definition of pm_q_prop,
Using lemma plain_memory_read_write_res[State, Data],
Installing automatic rewrites from: lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_write_q_data_stmt`.
Q.E.D.

C.179.8 Plain_Mem_Rewrites5.pm_q_prop_single_write

Terse proof for `pm_q_prop_single_write`.

`pm_q_prop_single_write`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm`states(s) ⊃
pm_q_prop(pm)(write_data(pm, dt)(addr, data)(s))

Expanding the definition of pm_q_prop,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_write_data_ok[State, Data],
Using lemma plain_memory_transformer_invariant_write_data[State, Data],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma expr_transformer_invariant_next_ok,
Expanding the definition of singleton,
Expanding the definition of has_next_state,
which is trivially true.
This completes the proof of `pm_q_prop_single_write`.
Q.E.D.

C.179.9 Plain_Mem_Rewrites5.pm_q_prop_read_write_single

Terse proof for `pm_q_prop_read_write_single`.

`pm_q_prop_read_write_single`:

{1} ∀ (addr: Address, data: Data, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧ in_blessed_memory?(dt, addr, pm`rw_addr) ∧ pm`states(s) ⊃
pm_q_prop_read(pm, dt, addr)(write_data(pm, dt)(addr, data)(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma pm_q_prop_single_write,
Using lemma plain_memory_read_write_ok[State, Data],
Expanding the definition of pm_q_prop,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ##,
Expanding the definition of ##,
which is trivially true.
This completes the proof of `pm_q_prop_read_write_single`.
Q.E.D.

C.179.10 Plain_Mem_Rewrites5.pm_q_prop_e2s

Terse proof for `pm_q_prop_e2s`.

`pm_q_prop_e2s`:

{1} ∀ (expr: [State → ExprResult[State, Data]], pm: Plain_Memory[State], s: State):
pm_q_prop(pm)(e2s(expr)(s)) = pm_q_prop(pm)(expr(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_e2s`.
Q.E.D.

C.179.11 Plain_Mem_Rewrites5.pm_q_prop_stmt_e2s

Terse proof for `pm_q_prop_stmt_e2s`.
`pm_q_prop_stmt_e2s`:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], pm: Plain_Memory[State], s: State,
          stmt: [State → StmtResult[State]]):
       pm_q_prop(pm)(stmt(s)) ⊃
        pm_q_prop(pm)((stmt ## e2s(expr))(s)) = pm_q_prop(pm)((stmt ## expr)(s))
```

Expanding the definition of pm_q_prop,
Repeatedly Skolemizing and flattening,
Using lemma pm_q_prop_e2s,
Installing automatic rewrites from: ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_q_prop_stmt_e2s`.
Q.E.D.

C.180 Proofs for Plain_Mem_Rewrites6 (plain_memory_rewrites.pvs)

C.180.1 Plain_Mem_Rewrites6.pm_q_prop_read_write_other_q_expr

Terse proof for `pm_q_prop_read_write_other_q_expr`.
`pm_q_prop_read_write_other_q_expr`:

```
{1}  ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧
        blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
         in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
          in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
           pm_q_prop_read(pm, dt1, addr1)(q(s))
        ⊃ pm_q_prop_read(pm, dt1, addr1)((q ## write_data(pm, dt2)(addr2, data2))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma pm_q_prop_single_write[State, Data2],
Expanding the definition of pm_q_prop,
Using lemma plain_memory_read_write_other_ok[State, Data2, Data1],
Installing automatic rewrites from: ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:

`pm_q_prop_read_write_other_q_expr.1:`

{-1}	plain_memory?(pm′)
{-2}	pm′‘states(state(q′(s′)))
{-3}	in_blessed_memory?(dt2′, addr2′, pm′‘rw_addr)
{-4}	in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}	OK?(write_data(pm′, dt2′)(addr2′, data2′)(state(q′(s′))))
{-6}	pm′‘states(state(write_data(pm′, dt2′)(addr2′, data2′)(state(q′(s′)))))
{-7}	interpreted_data_type?⌈Data1⌉(dt1′)
{-8}	interpreted_data_type?⌈Data2⌉(dt2′)
{-9}	blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-10}	OK?(q′(s′))
{-11}	OK?(read_data(pm′, dt1′)(addr1′)(state(q′(s′))))
{1}	valid_in_mem(pm′, dt1′)(addr1′)(state(q′(s′)))
{2}	OK?(read_data(pm′, dt1′)(addr1′) (state(write_data(pm′, dt2′)(addr2′, data2′)(state(q′(s′))))))

Expanding the definition of valid_in_mem,

Expanding the definition of read_data,

Rewriting using read_data_valid_in_mem, matching in *,

This completes the proof of `pm_q_prop_read_write_other_q_expr.1`.

`pm_q_prop_read_write_other_q_expr.2:`

{-1}	plain_memory?(pm′)
{-2}	pm′‘states(state(q′(s′)))
{-3}	in_blessed_memory?(dt2′, addr2′, pm′‘rw_addr)
{-4}	in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}	OK?(write_data(pm′, dt2′)(addr2′, data2′)(state(q′(s′))))
{-6}	pm′‘states(state(write_data(pm′, dt2′)(addr2′, data2′)(state(q′(s′)))))
{-7}	interpreted_data_type?⌈Data1⌉(dt1′)
{-8}	interpreted_data_type?⌈Data2⌉(dt2′)
{-9}	blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-10}	OK?(q′(s′))
{-11}	OK?(read_data(pm′, dt1′)(addr1′)(state(q′(s′))))
{1}	blocks_disjoint?(addr2′, size(uidt(dt2′)), addr1′, size(uidt(dt1′)))
{2}	OK?(read_data(pm′, dt1′)(addr1′) (state(write_data(pm′, dt2′)(addr2′, data2′)(state(q′(s′))))))

Keeping (-9 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_q_prop_read_write_other_q_expr.2`.

Q.E.D.

C.180.2 Plain_Mem_Rewrites6.pm_q_prop_read_read_q_expr

Terse proof for `pm_q_prop_read_read_q_expr`.

`pm_q_prop_read_read_q_expr:`

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
q: [State → ExprResult[State, Data_q]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
in_blessed_memory?(dt2, addr2, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
pm_q_prop_read(pm, dt1, addr1)($q(s)$) ∧ pm_q_prop_read(pm, dt2, addr2)($q(s)$)
⊃ pm_q_prop_read(pm, dt2, addr2)((q ## read_data(pm, dt1)(addr1))(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Applying disjunctive simplification to flatten sequent,
Using lemma pm_q_prop_read_read_single[State, Data1, Data2],
Expanding the definition of read_data,
Expanding the definition of valid_in_mem,
Installing automatic rewrites from: ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`pm_q_prop_read_read_q_expr.1:`

{-1} plain_memory?(pm′)
{-2} in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-3} in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} pm′‘states(state($q'(s')$))
{-5} interpreted_data_type?[Data1](dt1′)
{-6} interpreted_data_type?[Data2](dt2′)
{-7} OK?($q'(s')$)
{-8} OK?(read_data(pm′‘mem, dt1′)(addr1′)(state($q'(s')$)))
{-9} OK?(read_data(pm′‘mem, dt2′)(addr2′)(state($q'(s')$)))

{1} valid_in_mem(pm′‘mem, dt2′)(addr2′)(state($q'(s')$))
{2} pm_q_prop_read(pm′, dt2′, addr2′)
(read_data(pm′‘mem, dt1′)(addr1′)(state($q'(s')$)))

Rewriting using read_data_valid_in_mem, matching in *,
This completes the proof of `pm_q_prop_read_read_q_expr.1`.
`pm_q_prop_read_read_q_expr.2:`

{-1} plain_memory?(pm′)
{-2} in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-3} in_blessed_memory?(dt2′, addr2′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-4} pm′‘states(state($q'(s')$))
{-5} interpreted_data_type?[Data1](dt1′)
{-6} interpreted_data_type?[Data2](dt2′)
{-7} OK?($q'(s')$)
{-8} OK?(read_data(pm′‘mem, dt1′)(addr1′)(state($q'(s')$)))
{-9} OK?(read_data(pm′‘mem, dt2′)(addr2′)(state($q'(s')$)))

{1} valid_in_mem(pm′‘mem, dt1′)(addr1′)(state($q'(s')$))
{2} pm_q_prop_read(pm′, dt2′, addr2′)
(read_data(pm′‘mem, dt1′)(addr1′)(state($q'(s')$)))

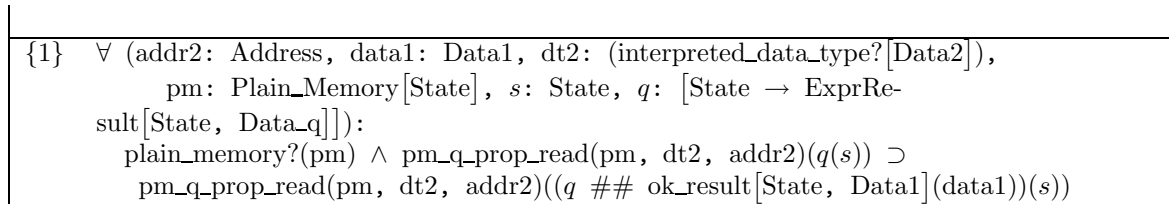
Rewriting using read_data_valid_in_mem, matching in *,
This completes the proof of `pm_q_prop_read_read_q_expr.2`.

Q.E.D.

C.180.3 Plain_Mem_Rewrites6.pm_q_prop_read_ok_result_q_expr

Terse proof for `pm_q_prop_read_ok_result_q_expr`.

`pm_q_prop_read_ok_result_q_expr`:

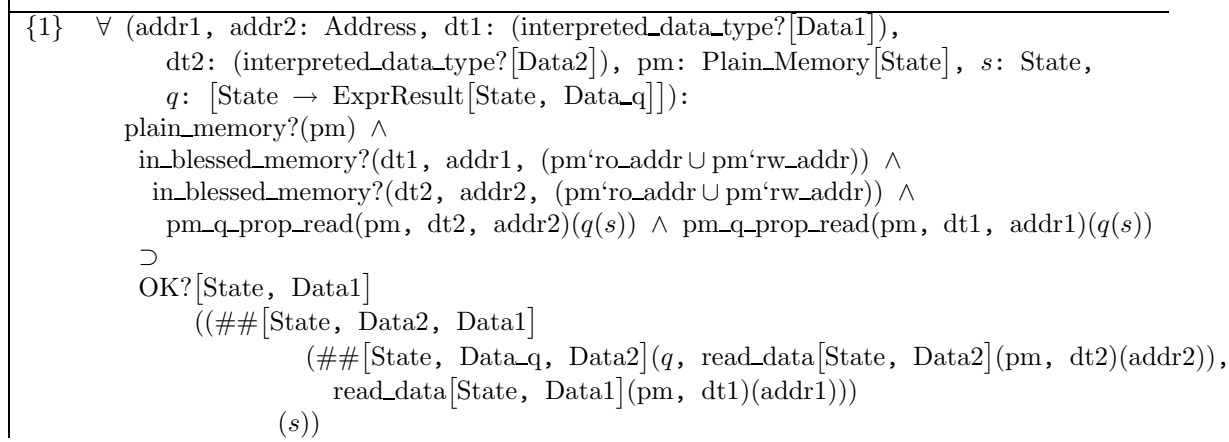


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_read_ok_result_q_expr`.
Q.E.D.

C.180.4 Plain_Mem_Rewrites6.plain_memory_read_read_q_data_expr_TCC1

Terse proof for `plain_memory_read_read_q_data_expr_TCC1`.

`plain_memory_read_read_q_data_expr_TCC1`:



Expanding the definition of pm_q_prop_read,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_read_ok[State, Data2, Data1],
Installing automatic rewrites from: ## lift
Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_read_q_data_expr_TCC1`.
Q.E.D.

C.180.5 Plain_Mem_Rewrites6.plain_memory_read_read_q_data_expr_TCC2

Terse proof for `plain_memory_read_read_q_data_expr_TCC2`.

plain_memory_read_read_q_data_expr_TCC2:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
         in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
          pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s))
        ⊃
        OK?[State, Data1]
            ((##[State, Data_q, Data1](q, read_data[State, Data1](pm, dt1)(addr1)))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_read_read_q_data_expr_TCC2.
Q.E.D.

C.180.6 Plain_Mem_Rewrites6.plain_memory_read_read_q_data_expr

Terse proof for plain_memory_read_read_q_data_expr.

plain_memory_read_read_q_data_expr:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
         in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
          pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s))
        ⊃
        data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) =
         data((q ## read_data(pm, dt1)(addr1))(s))
```

Expanding the definition of pm_q_prop_read,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_read[State, Data2, Data1],
Installing automatic rewrites from: ## lift valid_in_mem read_data read_data_valid_in_mem
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_read_read_q_data_expr.
Q.E.D.

C.181 Proofs for Plain_Mem_Rewrites7 (plain_memory_rewrites.pvs)

C.181.1 Plain_Mem_Rewrites7.pm_q_prop_read_write_other_q_stmt

Terse proof for pm_q_prop_read_write_other_q_stmt.

`pm_q_prop_read_write_other_q_stmt:`

{1} ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]):
plain_memory?(pm) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
in_blessed_memory?(dt1, addr1, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
pm_q_prop_read(pm, dt1, addr1)($q(s)$)
⊃ pm_q_prop_read(pm, dt1, addr1)((q ## write_data(pm, dt2)(addr2, data2))(s))

Repeatedly Skolemizing and flattening,

Expanding the definition of pm_q_prop_read,

Using lemma pm_q_prop_single_write[State, Data2],

Expanding the definition of pm_q_prop,

Using lemma plain_memory_read_write_other_ok[State, Data2, Data1],

Installing automatic rewrites from: ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:

`pm_q_prop_read_write_other_q_stmt.1:`

{-1} plain_memory?(pm′)
{-2} pm′‘states(state($q'(s')$))
{-3} in_blessed_memory?(dt2′, addr2′, pm′‘rw_addr)
{-4} in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5} OK?(write_data(pm′, dt2′)(addr2′, data2′)(state($q'(s')$)))
{-6} pm′‘states(state(write_data(pm′, dt2′)(addr2′, data2′)(state($q'(s')$))))
{-7} interpreted_data_type?[Data1](dt1′)
{-8} interpreted_data_type?[Data2](dt2′)
{-9} blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-10} OK?($q'(s')$)
{-11} OK?(read_data(pm′, dt1′)(addr1′)(state($q'(s')$)))

{1} valid_in_mem(pm′, dt1′)(addr1′)(state($q'(s')$))
{2} OK?(read_data(pm′, dt1′)(addr1′)
(state(write_data(pm′, dt2′)(addr2′, data2′)(state($q'(s')$)))))

Expanding the definition of valid_in_mem,

Expanding the definition of read_data,

Rewriting using read_data_valid_in_mem, matching in *,

This completes the proof of `pm_q_prop_read_write_other_q_stmt.1`.

`pm_q_prop_read_write_other_q_stmt.2:`

{-1}	plain_memory?(pm′)
{-2}	pm′‘states(state($q'(s')$))
{-3}	in_blessed_memory?(dt2′, addr2′, pm′‘rw_addr)
{-4}	in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}	OK?(write_data(pm′, dt2′)(addr2′, data2′)(state($q'(s')$)))
{-6}	pm′‘states(state(write_data(pm′, dt2′)(addr2′, data2′)(state($q'(s')$))))
{-7}	interpreted_data_type?[Data1](dt1′)
{-8}	interpreted_data_type?[Data2](dt2′)
{-9}	blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-10}	OK?($q'(s')$)
{-11}	OK?(read_data(pm′, dt1′)(addr1′)(state($q'(s')$)))
{1}	blocks_disjoint?(addr2′, size(uidt(dt2′)), addr1′, size(uidt(dt1′)))
{2}	OK?(read_data(pm′, dt1′)(addr1′) (state(write_data(pm′, dt2′)(addr2′, data2′)(state($q'(s')$)))))

Keeping (-9 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_q_prop_read_write_other_q_stmt.2`.

Q.E.D.

C.181.2 Plain_Mem_Rewrites7.plain_memory_read_write_other_single_data_TCC1

Terse proof for `plain_memory_read_write_other_single_data_TCC1`.

`plain_memory_read_write_other_single_data_TCC1:`

{1}	∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]), dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State): pm‘states(s) ∧ pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) ∧ blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧ in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧ in_blessed_memory?(dt1, addr1, (pm‘ro_addr ∪ pm‘rw_addr)) ∧ plain_memory?(pm) ⊃ OK?[State, Data1] ((##[State, Unit, Data1] (write_data[State, Data2](pm, dt2)(addr2, data2), read_data[State, Data1](pm, dt1)(addr1))) (s))

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_read_write_other_ok[State, Data2, Data1], matching in *,

we get 2 subgoals:

`plain_memory_read_write_other_single_data_TCC1.1`:

```
{-1}  interpreted_data_type?[Data1](dt1′)
{-2}  interpreted_data_type?[Data2](dt2′)
{-3}  pm′`states(s′)
{-4}  pm_q_prop(pm′)(read_data(pm′, dt1′)(addr1′)(s′))
{-5}  blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-6}  in_blessed_memory?(dt2′, addr2′, pm′`rw_addr)
{-7}  in_blessed_memory?(dt1′, addr1′, (pm′`ro_addr ∪ pm′`rw_addr))
{-8}  plain_memory?(pm′)
-------------------------------------------------------------------------
{1}   blocks_disjoint?(addr2′, size(uidt(dt2′)), addr1′, size(uidt(dt1′)))
{2}   OK?[State, Data1]
          ((##[State, Unit, Data1]
                (write_data[State, Data2](pm′, dt2′)(addr2′, data2′),
                  read_data[State, Data1](pm′, dt1′)(addr1′)))
              (s′))
```

Keeping (-5 1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_memory_read_write_other_single_data_TCC1.1`.

`plain_memory_read_write_other_single_data_TCC1.2`:

```
{-1}  interpreted_data_type?[Data1](dt1′)
{-2}  interpreted_data_type?[Data2](dt2′)
{-3}  pm′`states(s′)
{-4}  pm_q_prop(pm′)(read_data(pm′, dt1′)(addr1′)(s′))
{-5}  blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-6}  in_blessed_memory?(dt2′, addr2′, pm′`rw_addr)
{-7}  in_blessed_memory?(dt1′, addr1′, (pm′`ro_addr ∪ pm′`rw_addr))
{-8}  plain_memory?(pm′)
-------------------------------------------------------------------------
{1}   valid_in_mem(pm′, dt1′)(addr1′)(s′)
{2}   OK?[State, Data1]
          ((##[State, Unit, Data1]
                (write_data[State, Data2](pm′, dt2′)(addr2′, data2′),
                  read_data[State, Data1](pm′, dt1′)(addr1′)))
              (s′))
```

Keeping (-4 1) and hiding *,

Expanding the definition of pm_q_prop,

Installing automatic rewrites from: read_data valid_in_mem read_data_valid_in_mem

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_read_write_other_single_data_TCC1.2`.

Q.E.D.

C.181.3 Plain_Mem_Rewrites7.plain_memory_read_write_other_single_data_TCC2

Terse proof for `plain_memory_read_write_other_single_data_TCC2`.

plain_memory_read_write_other_single_data_TCC2:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       pm‘states(s) ∧
        pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) ∧
         blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
          in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
           in_blessed_memory?(dt1, addr1, (pm‘ro_addr ∪ pm‘rw_addr)) ∧ plain_memory?(pm)
       ⊃ OK?[State, Data1](read_data[State, Data1](pm, dt1)(addr1)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_read_write_other_single_data_TCC2.
Q.E.D.

C.181.4 Plain_Mem_Rewrites7.plain_memory_read_write_other_single_data

Terse proof for plain_memory_read_write_other_single_data.

plain_memory_read_write_other_single_data:

```
{1}  ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, (pm‘ro_addr ∪ pm‘rw_addr)) ∧
         in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           pm_q_prop(pm)(read_data(pm, dt1)(addr1)(s)) ∧ pm‘states(s)
        ⊃
        data((write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s)) =
         data(read_data(pm, dt1)(addr1)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop,
Rewriting using plain_memory_read_write_other_res, matching in *,
we get 2 subgoals:

plain_memory_read_write_other_single_data.1:

```
{-1}  interpreted_data_type?[Data1](dt1′)
{-2}  interpreted_data_type?[Data2](dt2′)
{-3}  plain_memory?(pm′)
{-4}  in_blessed_memory?(dt1′, addr1′, (pm′‘ro_addr ∪ pm′‘rw_addr))
{-5}  in_blessed_memory?(dt2′, addr2′, pm′‘rw_addr)
{-6}  blocks_disjoint?(addr1′, size(uidt(dt1′)), addr2′, size(uidt(dt2′)))
{-7}  OK?(read_data(pm′, dt1′)(addr1′)(s′)) ∧
       pm′‘states(state(read_data(pm′, dt1′)(addr1′)(s′)))
{-8}  pm′‘states(s′)
{1}   blocks_disjoint?(addr2′, size(uidt(dt2′)), addr1′, size(uidt(dt1′)))
{2}   data((write_data(pm′, dt2′)(addr2′, data2′) ## read_data(pm′, dt1′)(addr1′))
                (s′))
       = data(read_data(pm′, dt1′)(addr1′)(s′))
```

Keeping (-6 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_read_write_other_single_data.1`.
`plain_memory_read_write_other_single_data.2`:

```
{-1}  interpreted_data_type?[Data1](dt1')
{-2}  interpreted_data_type?[Data2](dt2')
{-3}  plain_memory?(pm')
{-4}  in_blessed_memory?(dt1', addr1', (pm'`ro_addr ∪ pm'`rw_addr))
{-5}  in_blessed_memory?(dt2', addr2', pm'`rw_addr)
{-6}  blocks_disjoint?(addr1', size(uidt(dt1')), addr2', size(uidt(dt2')))
{-7}  OK?(read_data(pm', dt1')(addr1')(s')) ∧
        pm'`states(state(read_data(pm', dt1')(addr1')(s')))
{-8}  pm'`states(s')
-------------------------------------------------------------------------
{1}   valid_in_mem(pm', dt1')(addr1')(s')
{2}   data((write_data(pm', dt2')(addr2', data2') ## read_data(pm', dt1')(addr1'))
                (s'))
        = data(read_data(pm', dt1')(addr1')(s'))
```

Expanding the definition of valid_in_mem,
Expanding the definition of read_data,
Rewriting using read_data_valid_in_mem, matching in *,
This completes the proof of `plain_memory_read_write_other_single_data.2`.
Q.E.D.

C.181.5 Plain_Mem_Rewrites7.pm_q_prop_read_read_q_stmt

Terse proof for `pm_q_prop_read_read_q_stmt`.
`pm_q_prop_read_read_q_stmt`:

```
-------------------------------------------------------------------------
{1}   ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
           dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
           q: [State → StmtResult[State]]):
        plain_memory?(pm) ∧
          in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
            in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
              pm_q_prop_read(pm, dt1, addr1)(q(s)) ∧ pm_q_prop_read(pm, dt2, addr2)(q(s))
          ⊃ pm_q_prop_read(pm, dt2, addr2)((q ## read_data(pm, dt1)(addr1))(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Applying disjunctive simplification to flatten sequent,
Using lemma pm_q_prop_read_read_single[State, Data1, Data2],
Installing automatic rewrites from: ## lift read_data valid_in_mem read_data_valid_in_mem
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `pm_q_prop_read_read_q_stmt`.
Q.E.D.

C.181.6 Plain_Mem_Rewrites7.pm_q_prop_read_ok_result_q_stmt

Terse proof for `pm_q_prop_read_ok_result_q_stmt`.

pm_q_prop_read_ok_result_q_stmt:

```
{1}  ∀ (addr2: Address, data1: Data1, dt2: (interpreted_data_type?[Data2]),
          pm: Plain_Memory[State], s: State, q: [State → StmtResult[State]]):
        plain_memory?(pm) ∧ pm_q_prop_read(pm, dt2, addr2)(q(s)) ⊃
         pm_q_prop_read(pm, dt2, addr2)((q ## ok_result[State, Data1](data1))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of pm_q_prop_read_ok_result_q_stmt.

Q.E.D.

C.181.7 Plain_Mem_Rewrites7.plain_memory_read_write_other_q_data_stmt_TCC1

Terse proof for plain_memory_read_write_other_q_data_stmt_TCC1.

plain_memory_read_write_other_q_data_stmt_TCC1:

```
{1}  ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → StmtResult[State]]):
        plain_memory?(pm) ∧
         in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
          in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
           blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
            pm_q_prop_read(pm, dt1, addr1)(q(s))
         ⊃
         OK?[State, Data1]
             ((##[State, Unit, Data1]
                    (##[State, Unit](q, write_data[State, Data2](pm, dt2)(addr2, data2)),
                      read_data[State, Data1](pm, dt1)(addr1)))
                (s))
```

Repeatedly Skolemizing and flattening,

Using lemma pm_q_prop_read_write_other_q_stmt,

Expanding the definition of pm_q_prop_read,

Installing automatic rewrites from: ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of plain_memory_read_write_other_q_data_stmt_TCC1.

Q.E.D.

C.181.8 Plain_Mem_Rewrites7.plain_memory_read_write_other_q_data_stmt_TCC2

Terse proof for plain_memory_read_write_other_q_data_stmt_TCC2.

`plain_memory_read_write_other_q_data_stmt_TCC2`:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
q: [State → StmtResult[State]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
pm_q_prop_read(pm, dt1, addr1)(q(s))
⊃
OK?[State, Data1]
((##[State, Data1](q, read_data[State, Data1](pm, dt1)(addr1)))(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_read_write_other_q_data_stmt_TCC2`.
Q.E.D.

C.181.9 Plain_Mem_Rewrites7.plain_memory_read_write_other_q_data_stmt

Terse proof for `plain_memory_read_write_other_q_data_stmt`.
`plain_memory_read_write_other_q_data_stmt`:

{1} ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
q: [State → StmtResult[State]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
pm_q_prop_read(pm, dt1, addr1)(q(s))
⊃
data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s))
= data((q ## read_data(pm, dt1)(addr1))(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma plain_memory_read_write_other_single_data,
Using lemma pm_q_prop_single_read[State, Data1],
Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_write_other_q_data_stmt`.
Q.E.D.

C.181.10 Plain_Mem_Rewrites7.plain_memory_read_read_q_data_stmt_TCC1

Terse proof for `plain_memory_read_read_q_data_stmt_TCC1`.

plain_memory_read_read_q_data_stmt_TCC1:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → StmtResult[State]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
         in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
          pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s))
        ⊃
        OK?[State, Data1]
            ((##[State, Data2, Data1]
                    (##[State, Data2](q, read_data[State, Data2](pm, dt2)(addr2)),
                      read_data[State, Data1](pm, dt1)(addr1)))
                  (s))
```

Expanding the definition of pm_q_prop_read,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_read_ok[State, Data2, Data1],
Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem ## lift
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_read_read_q_data_stmt_TCC1.
Q.E.D.

C.181.11 Plain_Mem_Rewrites7.plain_memory_read_read_q_data_stmt_TCC2

Terse proof for plain_memory_read_read_q_data_stmt_TCC2.

plain_memory_read_read_q_data_stmt_TCC2:

```
{1}  ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → StmtResult[State]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
         in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
          pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s))
        ⊃
        OK?[State, Data1]
            ((##[State, Data1](q, read_data[State, Data1](pm, dt1)(addr1)))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_read_read_q_data_stmt_TCC2.
Q.E.D.

C.181.12 Plain_Mem_Rewrites7.plain_memory_read_read_q_data_stmt

Terse proof for plain_memory_read_read_q_data_stmt.

`plain_memory_read_read_q_data_stmt`:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
q: [State → StmtResult[State]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt2, addr2, (pm`ro_addr ∪ pm`rw_addr)) ∧
pm_q_prop_read(pm, dt2, addr2)(q(s)) ∧ pm_q_prop_read(pm, dt1, addr1)(q(s))
⊃
data((q ## read_data(pm, dt2)(addr2) ## read_data(pm, dt1)(addr1))(s)) =
data((q ## read_data(pm, dt1)(addr1))(s))

Expanding the definition of pm_q_prop_read,
Repeatedly Skolemizing and flattening,
Using lemma plain_memory_read_read[State, Data2, Data1],
Installing automatic rewrites from: ## lift valid_in_mem read_data read_data_valid_in_mem
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_read_q_data_stmt`.
Q.E.D.

C.181.13 Plain_Mem_Rewrites7.pm_q_prop_read_e2s

Terse proof for `pm_q_prop_read_e2s`.

`pm_q_prop_read_e2s`:

{1} ∀ (addr1: Address, dt1: (interpreted_data_type?[Data1]),
expr1: [State → ExprResult[State, Data2]], pm: Plain_Memory[State], s: State):
pm_q_prop_read(pm, dt1, addr1)(expr1(s)) ⊃
pm_q_prop_read(pm, dt1, addr1)(e2s(expr1)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_read_e2s`.
Q.E.D.

C.181.14 Plain_Mem_Rewrites7.pm_q_prop_read_stmt_e2s

Terse proof for `pm_q_prop_read_stmt_e2s`.

`pm_q_prop_read_stmt_e2s`:

{1} ∀ (addr1: Address, dt1: (interpreted_data_type?[Data1]),
expr1: [State → ExprResult[State, Data2]], pm: Plain_Memory[State], s: State,
stmt: [State → StmtResult[State]]):
pm_q_prop_read(pm, dt1, addr1)(stmt(s)) ⊃
pm_q_prop_read(pm, dt1, addr1)((stmt ## e2s(expr1))(s)) =
pm_q_prop_read(pm, dt1, addr1)((stmt ## expr1)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `pm_q_prop_read_stmt_e2s`.
Q.E.D.

C.182 Proofs for Plain_Mem_Rewrites8 (plain_memory_rewrites.pvs)

C.182.1 Plain_Mem_Rewrites8.plain_memory_read_write_other_q_data_expr_TCC1

Terse proof for `plain_memory_read_write_other_q_data_expr_TCC1`.

`plain_memory_read_write_other_q_data_expr_TCC1:`

```
{1}  ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
          q: [State → ExprResult[State, Data_q]]):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
         in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           pm_q_prop_read(pm, dt1, addr1)(q(s))
        ⊃
        OK?[State, Data1]
            ((##[State, Unit, Data1]
                  (##[State, Data_q, Unit]
                        (q, write_data[State, Data2](pm, dt2)(addr2, data2)),
                     read_data[State, Data1](pm, dt1)(addr1)))
                (s))
```

Repeatedly Skolemizing and flattening,

Using lemma pm_q_prop_read_write_other_q_expr[State, Data1, Data2, Data_q],

Expanding the definition of pm_q_prop_read,

Installing automatic rewrites from: ## lift

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_read_write_other_q_data_expr_TCC1`.

Q.E.D.

C.182.2 Plain_Mem_Rewrites8.plain_memory_read_write_other_q_data_expr_TCC2

Terse proof for `plain_memory_read_write_other_q_data_expr_TCC2`.

`plain_memory_read_write_other_q_data_expr_TCC2`:

{1} ∀ (addr1, addr2: Address, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
q: [State → ExprResult[State, Data_q]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
pm_q_prop_read(pm, dt1, addr1)(q(s))
⊃
OK?[State, Data1]
((##[State, Data_q, Data1](q, read_data[State, Data1](pm, dt1)(addr1)))(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_read_write_other_q_data_expr_TCC2`.
Q.E.D.

C.182.3 Plain_Mem_Rewrites8.plain_memory_read_write_other_q_data_expr

Terse proof for `plain_memory_read_write_other_q_data_expr`.

`plain_memory_read_write_other_q_data_expr`:

{1} ∀ (addr1, addr2: Address, data2: Data2, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State,
q: [State → ExprResult[State, Data_q]]):
plain_memory?(pm) ∧
in_blessed_memory?(dt1, addr1, (pm`ro_addr ∪ pm`rw_addr)) ∧
in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
pm_q_prop_read(pm, dt1, addr1)(q(s))
⊃
data((q ## write_data(pm, dt2)(addr2, data2) ## read_data(pm, dt1)(addr1))(s))
= data((q ## read_data(pm, dt1)(addr1))(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of pm_q_prop_read,
Using lemma plain_memory_read_write_other_single_data[State, Data1, Data2],
Expanding the definition of ##,
Expanding the definition of ##,
Using lemma pm_q_prop_single_read[State, Data1],
Installing automatic rewrites from: valid_in_mem read_data read_data_valid_in_mem
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_read_write_other_q_data_expr`.
Q.E.D.

C.183 Proofs for Plain_Memory (plain_memory.pvs)

C.183.1 Plain_Memory.plain_memory_invariant_read_transformers_ro_addr

Terse proof for `plain_memory_invariant_read_transformers_ro_addr`.

`plain_memory_invariant_read_transformers_ro_addr:`

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
        transformer_invariant?(pm`states, memory_read_transformers(pm`mem, pm`ro_addr))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-1 1) and hiding *,
Using lemma unchanged_memory_invariant_invariant,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -2,
Using lemma transformer_invariant_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_invariant_read_transformers_ro_addr`.
Q.E.D.

C.183.2 Plain_Memory.plain_memory_invariant_read_transformers_rw_addr

Terse proof for `plain_memory_invariant_read_transformers_rw_addr`.

`plain_memory_invariant_read_transformers_rw_addr:`

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
        transformer_invariant?(pm`states, memory_read_transformers(pm`mem, pm`rw_addr))
```

Skolemizing and flattening,
Expanding the definition of plain_memory?,
Expanding the definition of changed_memory_invariant?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_invariant_read_transformers_rw_addr`.
Q.E.D.

C.183.3 Plain_Memory.plain_memory_invariant_read_transformers_ro_rw_addr

Terse proof for `plain_memory_invariant_read_transformers_ro_rw_addr`.

plain_memory_invariant_read_transformers_ro_rw_addr:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
         transformer_invariant?(pm‘states,
                                memory_read_transformers(pm‘mem, (pm‘ro_addr ∪ pm‘rw_addr)))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_invariant_read_transformers_ro_addr,
Using lemma plain_memory_invariant_read_transformers_rw_addr,
Simplifying, rewriting, and recording with decision procedures,
Using lemma transformer_invariant_union_transformers,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using memory_read_transformers_union, matching in *,
This completes the proof of plain_memory_invariant_read_transformers_ro_rw_addr.
Q.E.D.

C.183.4 Plain_Memory.plain_memory_invariant_write_transformers_rw_addr

Terse proof for plain_memory_invariant_write_transformers_rw_addr.

plain_memory_invariant_write_transformers_rw_addr:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
         transformer_invariant?(pm‘states, memory_write_transformers(pm‘mem, pm‘rw_addr))
```

Skolemizing and flattening,
Expanding the definition of plain_memory?,
Expanding the definition of changed_memory_invariant?,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of plain_memory_invariant_write_transformers_rw_addr.
Q.E.D.

C.183.5 Plain_Memory.plain_memory_invariant

Terse proof for plain_memory_invariant.

plain_memory_invariant:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
         transformer_invariant?(pm‘states,
                                (pm‘other_actions ∪ ((memory_read_transformers(pm‘mem,  (pm‘ro_addr ∪ pm‘
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Expanding the definition of unchanged_memory_invariant?,
Applying disjunctive simplification to flatten sequent,
Keeping (-1 -3 1) and hiding *,
Using lemma transformer_invariant_union_transformers,

Case splitting on union(pm!1‘other_actions, union(union(memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr)), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr)))) = union(memory_write_transformers(pm!1‘mem, pm!1‘rw_addr), union(union(pm!1‘other_actions, memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr))), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr)))),
we get 2 subgoals:
`plain_memory_invariant.1:`

{-1} (pm′‘other_actions ∪ ((memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ∪ mem
=
(memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_actions ∪ memory_read_trans
{-2} transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr)) ∧
transformer_invariant?(pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_
⊃
transformer_invariant?(pm′‘states,
(memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_
{-3} transformer_invariant?(pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_a
{-4} transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))

{1} transformer_invariant?(pm′‘states,
(pm′‘other_actions ∪ ((memory_read_transformers(pm′‘mem, (pm′‘ro_

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_invariant.1`.
`plain_memory_invariant.2:`

{-1} transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr)) ∧
transformer_invariant?(pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_
⊃
transformer_invariant?(pm′‘states,
(memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_
{-2} transformer_invariant?(pm′‘states,
((pm′‘other_actions ∪ memory_read_transformers(pm′‘mem, (pm′‘ro_a
{-3} transformer_invariant?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))

{1} (pm′‘other_actions ∪ ((memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ∪ mem
=
(memory_write_transformers(pm′‘mem, pm′‘rw_addr) ∪ ((pm′‘other_actions ∪ memory_read_trans
{2} transformer_invariant?(pm′‘states,
(pm′‘other_actions ∪ ((memory_read_transformers(pm′‘mem, (pm′‘ro_

Keeping (1) and hiding *,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_invariant.2`.
Q.E.D.

C.183.6 Plain_Memory.plain_memory_unchanged_invariant

Terse proof for `plain_memory_unchanged_invariant`.

plain_memory_unchanged_invariant:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
         unchanged_memory_invariant?(pm‘mem, pm‘states,
                                     ((pm‘other_actions ∪ memory_read_transformers(pm‘mem,  (pm‘ro_addr ∪
                                     (pm‘ro_addr ∪ pm‘rw_addr))
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of plain_memory_unchanged_invariant.
Q.E.D.

C.183.7 Plain_Memory.plain_memory_unchanged_memory_invariant_write

Terse proof for plain_memory_unchanged_memory_invariant_write.
plain_memory_unchanged_memory_invariant_write:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
         unchanged_memory_invariant?(pm‘mem, pm‘states, mem-
       ory_write_transformers(pm‘mem, pm‘rw_addr),
                                     pm‘ro_addr)
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of plain_memory_unchanged_memory_invariant_write.
Q.E.D.

C.183.8 Plain_Memory.plain_memory_unchanged_memory_write_invariant

Terse proof for plain_memory_unchanged_memory_write_invariant.
plain_memory_unchanged_memory_write_invariant:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃ unchanged_memory_write_invariant?(pm‘mem, pm‘states, pm‘rw_addr)
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of plain_memory_unchanged_memory_write_invariant.
Q.E.D.

C.183.9 Plain_Memory.plain_memory_changed_memory_invariant

Terse proof for plain_memory_changed_memory_invariant.
plain_memory_changed_memory_invariant:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃ changed_memory_invariant?(pm‘mem, pm‘states, pm‘rw_addr)
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
This completes the proof of `plain_memory_changed_memory_invariant`.
Q.E.D.

C.183.10 Plain_Memory.plain_memory_invariant_read_block

Terse proof for `plain_memory_invariant_read_block`.
`plain_memory_invariant_read_block:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        transformer_invariant?(pm‘states,
                               memory_read_transformers(pm‘mem, ad-
     dress_block(addr, size)))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: memory_read_transformers_mono
Using lemma plain_memory_invariant_read_transformers_ro_rw_addr,
Simplifying, rewriting, and recording with decision procedures,
Using lemma transformer_invariant_mono_transformers,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_invariant_read_block`.
Q.E.D.

C.183.11 Plain_Memory.plain_memory_invariant_write_block

Terse proof for `plain_memory_invariant_write_block`.
`plain_memory_invariant_write_block:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
        transformer_invariant?(pm‘states,
                               memory_write_transformers(pm‘mem, ad-
     dress_block(addr, size)))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_invariant_write_transformers_rw_addr,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: memory_write_transformers_mono
Rewriting using transformer_invariant_mono_transformers, matching in * where transformers_1 gets memory_write_transformers(pm!1‘mem, address_block(addr!1, size!1)), transformers_2 gets memory_write_transformers(pm!1‘mem, pm!1‘rw_addr),
This completes the proof of `plain_memory_invariant_write_block`.
Q.E.D.

C.183.12 Plain_Memory.plain_memory_transformer_invariant_read_side_effects

Terse proof for `plain_memory_transformer_invariant_read_side_effects`.

plain_memory_transformer_invariant_read_side_effects:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        transformer_invariant?(pm‘states,
                               memory_read_side_effect_super_transformers(pm‘mem,
                                                                          address_block
                                                                          (addr, size)))
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Using lemma transformer_invariant_mono_transformers,
Using lemma unchanged_memory_invariant_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-9 1) and hiding *,
Rewriting using subset_transitive, matching in * where a gets memory_read_side_effect_super_transformers(pm!1‘mem, address_block (addr!1, size!1)), b gets memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), c gets union(union(pm!1‘other_actions, memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr))), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),
we get 2 subgoals:

plain_memory_transformer_invariant_read_side_effects.1:

```
{-1}  side_effect_content_unchanged(pm′‘rw_addr, pm′‘states, mem-
      ory_write_side_effect(pm′‘mem))
{1}   (memory_read_side_effect_super_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ memory_read_side_effe
{2}   (memory_read_side_effect_super_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ ((pm′‘other_actions ∪ r
```

Hiding formulas: 2,
Expanding the definition of subset?,
Expanding the definition of member,
Repeatedly Skolemizing and flattening,
Expanding the definition of memory_read_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -4 1) and hiding *,
Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets address_block(addr!1, size!1), c gets union(pm!1‘ro_addr, pm!1‘rw_addr),
This completes the proof of plain_memory_transformer_invariant_read_side_effects.1.

plain_memory_transformer_invariant_read_side_effects.2:

```
{-1}  side_effect_content_unchanged(pm′‘rw_addr, pm′‘states, mem-
      ory_write_side_effect(pm′‘mem))
{1}   (memory_read_side_effect_super_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)) ⊆ ((pm′‘other_actions ∪
{2}   (memory_read_side_effect_super_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ ((pm′‘other_actions ∪ r
```

Hiding formulas: (-1 2),
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_transformer_invariant_read_side_effects.2.
Q.E.D.

C.183.13 Plain_Memory.plain_memory_transformer_invariant_write_side_effects

Terse proof for `plain_memory_transformer_invariant_write_side_effects`.

`plain_memory_transformer_invariant_write_side_effects:`

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
transformer_invariant?(pm‘states,
memory_write_side_effect_super_transformers(pm‘mem, address_block
(addr, size)))

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Using lemma transformer_invariant_mono_transformers,
Using lemma unchanged_memory_invariant_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-9 1) and hiding *,
Rewriting using subset_transitive, matching in * where a gets memory_write_side_effect_super_transformers(pm!1‘address_block (addr!1, size!1)), b gets memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr), c gets union(union(pm!1‘other_actions, memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr))), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),
we get 2 subgoals:

`plain_memory_transformer_invariant_write_side_effects.1:`

{-1} side_effect_content_unchanged(pm′‘rw_addr, pm′‘states, memory_write_side_effect(pm′‘mem))
{1} (memory_write_side_effect_super_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ memory_w
{2} (memory_write_side_effect_super_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ ((pm′‘othe

Hiding formulas: 2,
Expanding the definition of subset?,
Expanding the definition of member,
Repeatedly Skolemizing and flattening,
Expanding the definition of memory_write_side_effect_super_transformers,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-2 -4 1) and hiding *,
Rewriting using subset_transitive, matching in * where a gets address_block(a!1, length(bl!1)), b gets address_block(addr!1, size!1), c gets pm!1‘rw_addr,
This completes the proof of `plain_memory_transformer_invariant_write_side_effects.1`.

`plain_memory_transformer_invariant_write_side_effects.2:`

{-1} side_effect_content_unchanged(pm′‘rw_addr, pm′‘states, memory_write_side_effect(pm′‘mem))
{1} (memory_write_side_effect_super_transformers(pm′‘mem, pm′‘rw_addr) ⊆ ((pm′‘other_actions ∪ me
{2} (memory_write_side_effect_super_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ ((pm′‘othe

Hiding formulas: (-1 2),

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_transformer_invariant_write_side_effects.2`.
Q.E.D.

C.183.14 Plain_Memory.plain_memory_transformer_invariant_read_list_block

Terse proof for `plain_memory_transformer_invariant_read_list_block`.
`plain_memory_transformer_invariant_read_list_block`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm`ro_addr ∪ pm`rw_addr)) ⊃
        transformer_invariant?(pm`states,
                               (memory_read_transformers(pm`mem, address_block(addr, size)) ∪ memory_re
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformer_invariant_read_side_effects,
Using lemma plain_memory_invariant_read_block,
Rewriting using transformer_invariant_union_transformers, matching in *,
This completes the proof of `plain_memory_transformer_invariant_read_list_block`.
Q.E.D.

C.183.15 Plain_Memory.plain_memory_transformer_invariant_write_list_block

Terse proof for `plain_memory_transformer_invariant_write_list_block`.
`plain_memory_transformer_invariant_write_list_block`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm`rw_addr) ⊃
        transformer_invariant?(pm`states,
                               (memory_write_transformers(pm`mem, address_block(addr, size)) ∪ memory_w
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_invariant_write_block,
Using lemma plain_memory_transformer_invariant_write_side_effects,
Rewriting using transformer_invariant_union_transformers, matching in *,
This completes the proof of `plain_memory_transformer_invariant_write_list_block`.
Q.E.D.

C.183.16 Plain_Memory.plain_memory_transformers_ok_read_ro_rw

Terse proof for `plain_memory_transformers_ok_read_ro_rw`.
`plain_memory_transformers_ok_read_ro_rw`:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
        transformers_ok?(pm`states, memory_read_transformers(pm`mem, (pm`ro_addr ∪ pm`rw_addr)))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: memory_read_transformers_mono

Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-5 1) and hiding *,
Using lemma transformers_ok_mono_transformers,
Simplifying, rewriting, and recording with decision procedures,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_transformers_ok_read_ro_rw`.
Q.E.D.

C.183.17 Plain_Memory.plain_memory_transformers_ok_read_block

Terse proof for `plain_memory_transformers_ok_read_block`.
`plain_memory_transformers_ok_read_block:`

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
transformers_ok?(pm‘states, memory_read_transformers(pm‘mem, address_block(addr, size)))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: memory_read_transformers_mono
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_read_transformers(pm!1‘mem, address_block(addr!1, size!1)), transformers_2 gets memory_read_transformers(pm!1‘mem, union(pm!1‘ro_addr, pm!1‘rw_addr)),
we get 2 subgoals:
`plain_memory_transformers_ok_read_block.1:`

{-1} size′ ≥ 0
{-2} plain_memory?(pm′)
{-3} (address_block(addr′, size′) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} (memory_read_transformers(pm′‘mem, address_block(addr′, size′)) ⊆ memory_read_transformers(p
{2} transformers_ok?(pm′‘states,
memory_read_transformers(pm′‘mem, address_block(addr′, size′)))

Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_transformers_ok_read_block.1`.
`plain_memory_transformers_ok_read_block.2:`

{-1} size′ ≥ 0
{-2} plain_memory?(pm′)
{-3} (address_block(addr′, size′) ⊆ (pm′‘ro_addr ∪ pm′‘rw_addr))

{1} transformers_ok?(pm′‘states,
memory_read_transformers(pm′‘mem, (pm′‘ro_addr ∪ pm′‘rw_addr)))
{2} transformers_ok?(pm′‘states,
memory_read_transformers(pm′‘mem, address_block(addr′, size′)))

Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-6 1) and hiding *,
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_memory_transformers_ok_read_block.2`.
Q.E.D.

C.183.18 Plain_Memory.plain_memory_transformers_ok_write_rw

Terse proof for `plain_memory_transformers_ok_write_rw`.
`plain_memory_transformers_ok_write_rw:`

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
        transformers_ok?(pm`states, memory_write_transformers(pm`mem, pm`rw_addr))
```

Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right
Skolemizing and flattening,
Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_write_transformers(pm!1`mem, pm!1`rw_addr), transformers_2 gets union(union(memory_read_transformers(pm!1`mem, union (pm!1`ro_addr, pm!1`rw_addr)), memory_write_transformers(pm!1`mem, pm!1`rw_addr)), union(memory_read_side_effect_super_transformers(pm!1`mem, union (pm!1`ro_addr, pm!1`rw_addr)), memory_write_side_effect_super_transformers(pm!1`mem, pm!1`rw_addr))),
This completes the proof of `plain_memory_transformers_ok_write_rw`.
Q.E.D.

C.183.19 Plain_Memory.plain_memory_transformers_ok_write_block

Terse proof for `plain_memory_transformers_ok_write_block`.
`plain_memory_transformers_ok_write_block:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm`rw_addr) ⊃
        transformers_ok?(pm`states, memory_write_transformers(pm`mem, ad-
     dress_block(addr, size)))
```

Skolemizing and flattening,
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_write_transformers(pm!1`mem, address_block(addr!1, size!1)), transformers_2 gets memory_write_transformers(pm!1`mem, pm!1`rw_addr),
we get 2 subgoals:
`plain_memory_transformers_ok_write_block.1:`

```
{-1}  plain_memory?(pm′)
{-2}  (address_block(addr′, size′) ⊆ pm′`rw_addr)
------------------------------------------------------------------------------
{1}   (memory_write_transformers(pm′`mem, address_block(addr′, size′)) ⊆ memory_write_transformers(pm′`mem,
{2}   transformers_ok?(pm′`states,
                       memory_write_transformers(pm′`mem, ad-
      dress_block(addr′, size′)))
```

Rewriting using memory_write_transformers_mono, matching in *,
This completes the proof of `plain_memory_transformers_ok_write_block.1`.

`plain_memory_transformers_ok_write_block.2:`

```
{-1}  plain_memory?(pm′)
{-2}  (address_block(addr′, size′) ⊆ pm′‘rw_addr)
------------------------------------------------
{1}   transformers_ok?(pm′‘states, memory_write_transformers(pm′‘mem, pm′‘rw_addr))
{2}   transformers_ok?(pm′‘states,
                       memory_write_transformers(pm′‘mem, ad-
      dress_block(addr′, size′)))
```

Rewriting using plain_memory_transformers_ok_write_rw, matching in *,
This completes the proof of `plain_memory_transformers_ok_write_block.2`.
Q.E.D.

C.183.20 Plain_Memory.plain_memory_transformers_ok_read_side_effects_ro_rw

Terse proof for `plain_memory_transformers_ok_read_side_effects_ro_rw`.
`plain_memory_transformers_ok_read_side_effects_ro_rw:`

```
------------------------------------------------
{1}   ∀ (pm: Plain_Memory):
        plain_memory?(pm) ⊃
         transformers_ok?(pm‘states,
                          memory_read_side_effect_super_transformers(pm‘mem,
                                                                     (pm‘ro_addr ∪ pm‘rw_
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Keeping (-5 1) and hiding *,
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), transformers_2 gets union(union(memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr)), union(memory_read_side_effect_super_transformers(pm!1‘
union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),
Keeping 1 and hiding *,
Rewriting using subset_bigger_union_right, matching in *,
Rewriting using union_subset1, matching in *,
This completes the proof of `plain_memory_transformers_ok_read_side_effects_ro_rw`.
Q.E.D.

C.183.21 Plain_Memory.plain_memory_transformers_ok_read_side_effects_block

Terse proof for `plain_memory_transformers_ok_read_side_effects_block`.
`plain_memory_transformers_ok_read_side_effects_block:`

```
------------------------------------------------
{1}   ∀ (addr: Address, pm: Plain_Memory, size: nat):
        plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
         transformers_ok?(pm‘states,
                          memory_read_side_effect_super_transformers(pm‘mem,
                                                                     ad-
      dress_block(addr, size)))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformers_ok_read_side_effects_ro_rw,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: memory_read_side_effect_super_transformers_mono
Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_transformers_ok_read_side_effects_block`.
Q.E.D.

C.183.22 Plain_Memory.plain_memory_transformers_ok_write_side_effects_ro_rw

Terse proof for `plain_memory_transformers_ok_write_side_effects_ro_rw`.
`plain_memory_transformers_ok_write_side_effects_ro_rw`:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
        transformers_ok?(pm‘states, memory_write_side_effect_super_transformers(pm‘mem, pm‘rw_addr))
```

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Keeping (-5 1) and hiding *,
Rewriting using transformers_ok_mono_transformers, matching in * where transformers_1 gets memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr), transformers_2 gets union(union(memory_read_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_transformers(pm!1‘mem, pm!1‘rw_addr)), union(memory_read_side_effect_super_transformers(pm!1‘mem, union (pm!1‘ro_addr, pm!1‘rw_addr)), memory_write_side_effect_super_transformers(pm!1‘mem, pm!1‘rw_addr))),
Keeping 1 and hiding *,
Rewriting using subset_bigger_union_right, matching in *,
Rewriting using union_subset3, matching in *,
This completes the proof of `plain_memory_transformers_ok_write_side_effects_ro_rw`.
Q.E.D.

C.183.23 Plain_Memory.plain_memory_transformers_ok_write_side_effects_block

Terse proof for `plain_memory_transformers_ok_write_side_effects_block`.
`plain_memory_transformers_ok_write_side_effects_block`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
        transformers_ok?(pm‘states,
                         memory_write_side_effect_super_transformers(pm‘mem,
                                                                     address_block(addr, size)))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformers_ok_write_side_effects_ro_rw,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: memory_write_side_effect_super_transformers_mono

Using lemma transformers_ok_mono_transformers,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_transformers_ok_write_side_effects_block`.
Q.E.D.

C.183.24 Plain_Memory.plain_memory_transformers_ok_read_list_block

Terse proof for `plain_memory_transformers_ok_read_list_block`.
`plain_memory_transformers_ok_read_list_block:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        transformers_ok?(pm‘states,
                         (memory_read_transformers(pm‘mem, address_block(addr, size)) ∪ memo
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformers_ok_read_block,
Using lemma plain_memory_transformers_ok_read_side_effects_block,
Rewriting using transformers_ok_union_transformers, matching in *,
This completes the proof of `plain_memory_transformers_ok_read_list_block`.
Q.E.D.

C.183.25 Plain_Memory.plain_memory_transformers_ok_write_list_block

Terse proof for `plain_memory_transformers_ok_write_list_block`.
`plain_memory_transformers_ok_write_list_block:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
        transformers_ok?(pm‘states,
                         (memory_write_transformers(pm‘mem, address_block(addr, size)) ∪ memo
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformers_ok_write_block,
Using lemma plain_memory_transformers_ok_write_side_effects_block,
Rewriting using transformers_ok_union_transformers, matching in *,
This completes the proof of `plain_memory_transformers_ok_write_list_block`.
Q.E.D.

C.183.26 Plain_Memory.plain_memory_unchanged_read_block_rw_addr

Terse proof for `plain_memory_unchanged_read_block_rw_addr`.

plain_memory_unchanged_read_block_rw_addr:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    memory_read_transformers(pm‘mem, ad-
     dress_block(addr, size)),
                                    pm‘rw_addr)
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_unchanged_invariant,
Using lemma unchanged_memory_invariant_mono,
Rewriting using union_commutative, matching in *,
Rewriting using union_subset1, matching in *,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-4 1) and hiding *,
Using lemma memory_read_transformers_mono,
Rewriting using subset_bigger_union_left, matching in * where a gets address_block(addr!1, size!1), b gets pm!1‘rw_addr, c gets pm!1‘ro_addr,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of plain_memory_unchanged_read_block_rw_addr.
Q.E.D.

C.183.27 Plain_Memory.plain_memory_unchanged_read_ro_rw_addr

Terse proof for plain_memory_unchanged_read_ro_rw_addr.

plain_memory_unchanged_read_ro_rw_addr:

```
{1}  ∀ (pm: Plain_Memory):
       plain_memory?(pm) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    memory_read_transformers(pm‘mem, (pm‘ro_addr ∪ pm‘rw_addr)),
                                    (pm‘ro_addr ∪ pm‘rw_addr))
```

Skolemizing and flattening,
Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left union_left union_right
Using lemma plain_memory_unchanged_invariant,
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_unchanged_read_ro_rw_addr.
Q.E.D.

C.183.28 Plain_Memory.plain_memory_unchanged_read_block_ro_rw_addr

Terse proof for plain_memory_unchanged_read_block_ro_rw_addr.

plain_memory_unchanged_read_block_ro_rw_addr:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    memory_read_transformers(pm‘mem, ad-
     dress_block(addr, size)),
                                    (pm‘ro_addr ∪ pm‘rw_addr))
```

Skolemizing and flattening,
Using lemma plain_memory_unchanged_read_ro_rw_addr,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Installing automatic rewrites from: subset_equal memory_read_transformers_mono
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of plain_memory_unchanged_read_block_ro_rw_addr.
Q.E.D.

C.183.29 Plain_Memory.plain_memory_unchanged_write_block_ro_addr

Terse proof for plain_memory_unchanged_write_block_ro_addr.

plain_memory_unchanged_write_block_ro_addr:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    memory_write_transformers(pm‘mem, ad-
     dress_block(addr, size)),
                                    pm‘ro_addr)
```

Repeatedly Skolemizing and flattening,
Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Using lemma unchanged_memory_invariant_mono,
Rewriting using subset_reflexive, matching in *,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using memory_write_transformers_mono[State], matching in *,
This completes the proof of plain_memory_unchanged_write_block_ro_addr.
Q.E.D.

C.183.30 Plain_Memory.plain_memory_unchanged_read_block

Terse proof for plain_memory_unchanged_read_block.

plain_memory_unchanged_read_block:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    memory_read_transformers(pm‘mem, ad-
     dress_block(addr, size)),
                                    address_block(addr, size))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_unchanged_read_block_ro_rw_addr,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Simplifying, rewriting, and recording with decision procedures,
Rewriting using subset_reflexive, matching in *,
This completes the proof of plain_memory_unchanged_read_block.
Q.E.D.

C.183.31 Plain_Memory.plain_memory_unchanged_read_side_effect_block_ro_rw

Terse proof for plain_memory_unchanged_read_side_effect_block_ro_rw.

plain_memory_unchanged_read_side_effect_block_ro_rw:

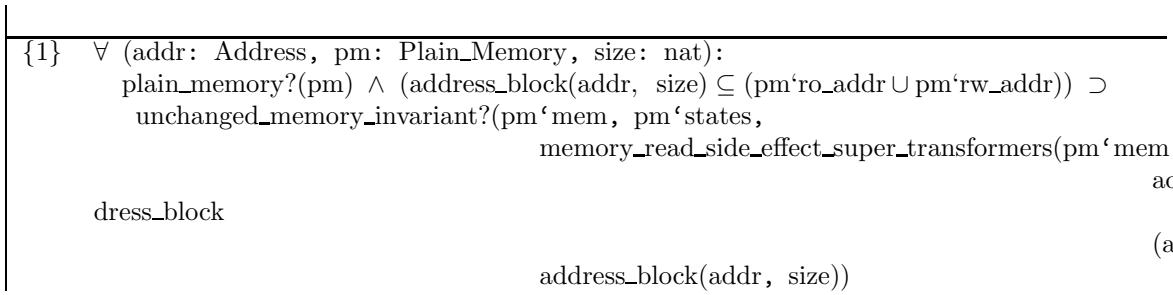
```
{1}  ∀ (addr: Address, pm: Plain_Memory, size: nat):
       plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
        unchanged_memory_invariant?(pm‘mem, pm‘states,
                                    memory_read_side_effect_super_transformers(pm‘mem,
                                                                               ad-
     dress_block
                                                                               (addr, size)),
                                    (pm‘ro_addr ∪ pm‘rw_addr))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_unchanged_invariant,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Installing automatic rewrites from: subset_equal subset_bigger_union_right subset_bigger_union_left memory_read_side_effect_super_transformers_mono
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_unchanged_read_side_effect_block_ro_rw.
Q.E.D.

C.183.32 Plain_Memory.plain_memory_unchanged_read_side_effect_block

Terse proof for plain_memory_unchanged_read_side_effect_block.

plain_memory_unchanged_read_side_effect_block:

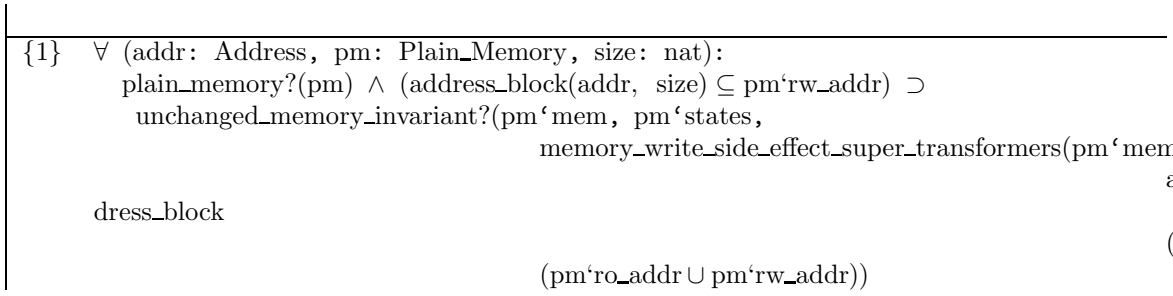


Repeatedly Skolemizing and flattening,
Using lemma plain_memory_unchanged_read_side_effect_block_ro_rw,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Installing automatic rewrites from: subset_equal
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_unchanged_read_side_effect_block.
Q.E.D.

C.183.33 Plain_Memory.plain_memory_unchanged_write_side_effect_block_ro_rw

Terse proof for plain_memory_unchanged_write_side_effect_block_ro_rw.

plain_memory_unchanged_write_side_effect_block_ro_rw:



Repeatedly Skolemizing and flattening,
Using lemma plain_memory_unchanged_invariant,
Simplifying, rewriting, and recording with decision procedures,
Using lemma unchanged_memory_invariant_mono,
Installing automatic rewrites from: subset_equal subset_bigger_union_right subset_bigger_union_left memory_write_side_effect_super_transformers_mono
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_unchanged_write_side_effect_block_ro_rw.
Q.E.D.

C.183.34 Plain_Memory.plain_memory_unchanged_write_side_effect_block

Terse proof for plain_memory_unchanged_write_side_effect_block.

plain_memory_unchanged_write_side_effect_block:

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
unchanged_memory_invariant?(pm‘mem, pm‘states,
memory_write_side_effect_super_transformers(pm‘mem,
address_block
(addr, size)),
address_block(addr, size))

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_unchanged_write_side_effect_block_ro_rw,
Simplifying, rewriting, and recording with decision procedures,
Installing automatic rewrites from: subset_equal union_subset1 union_subset3 union_left union_right subset_singleton subset_bigger_union_right subset_bigger_union_left
Using lemma unchanged_memory_invariant_mono,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_unchanged_write_side_effect_block.
Q.E.D.

C.183.35 Plain_Memory.plain_memory_unchanged_block

Terse proof for plain_memory_unchanged_block.
plain_memory_unchanged_block:

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
unchanged_memory_write_invariant?(pm‘mem, pm‘states, address_block(addr, size))

Repeatedly Skolemizing and flattening,
Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,
Keeping (-4 -9 1) and hiding *,
Rewriting using unchanged_memory_write_invariant_mono_addresses, matching in * where addresses_1 gets address_block(addr!1, size!1), addresses_2 gets pm!1‘rw_addr,
This completes the proof of plain_memory_unchanged_block.
Q.E.D.

C.183.36 Plain_Memory.plain_memory_changed_block

Terse proof for plain_memory_changed_block.
plain_memory_changed_block:

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
changed_memory_invariant?(pm‘mem, pm‘states, address_block(addr, size))

Repeatedly Skolemizing and flattening,
Expanding the definition of plain_memory?,
Applying disjunctive simplification to flatten sequent,

Rewriting using changed_memory_invariant_mono, matching in * where addresses_1 gets address_block(addr!1, size!1), addresses_2 gets pm!1‘rw_addr,
This completes the proof of `plain_memory_changed_block`.
Q.E.D.

C.183.37 Plain_Memory.plain_memory_side_effect_content_unchanged_read_block

Terse proof for `plain_memory_side_effect_content_unchanged_read_block`.
`plain_memory_side_effect_content_unchanged_read_block:`

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ (pm‘ro_addr ∪ pm‘rw_addr)) ⊃
side_effect_content_unchanged(address_block(addr, size), pm‘states,
memory_read_side_effect(pm‘mem))

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Keeping (-7 -9 1) and hiding *,
Rewriting using side_effect_content_unchanged_mono[State], matching in * where addresses_1 gets address_block(addr!1, size!1), addresses_2 gets union(pm!1‘ro_addr, pm!1‘rw_addr),
This completes the proof of `plain_memory_side_effect_content_unchanged_read_block`.
Q.E.D.

C.183.38 Plain_Memory.plain_memory_side_effect_content_unchanged_write_block

Terse proof for `plain_memory_side_effect_content_unchanged_write_block`.
`plain_memory_side_effect_content_unchanged_write_block:`

{1} ∀ (addr: Address, pm: Plain_Memory, size: nat):
plain_memory?(pm) ∧ (address_block(addr, size) ⊆ pm‘rw_addr) ⊃
side_effect_content_unchanged(address_block(addr, size), pm‘states,
memory_write_side_effect(pm‘mem))

Expanding the definition of plain_memory?,
Repeatedly Skolemizing and flattening,
Keeping (-8 -9 1) and hiding *,
Rewriting using side_effect_content_unchanged_mono[State], matching in * where addresses_1 gets address_block(addr!1, size!1), addresses_2 gets pm!1‘rw_addr,
This completes the proof of `plain_memory_side_effect_content_unchanged_write_block`.
Q.E.D.

C.183.39 Plain_Memory.plain_memory_memory_read_ok

Terse proof for `plain_memory_memory_read_ok`.
`plain_memory_memory_read_ok:`

{1} ∀ (addr: Address, pm: Plain_Memory, *s*: State):
plain_memory?(pm) ∧ pm‘states(*s*) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr) ⊃
OK?(memory_read(pm‘mem)(addr)(*s*))

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_transformers_ok_read_ro_rw,
Simplifying, rewriting, and recording with decision procedures,
Using lemma expr_transformers_ok_ok,
Installing automatic rewrites from: memory_read_transformers_memory_read
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `plain_memory_memory_read_ok`.
Q.E.D.

C.183.40 Plain_Memory.plain_memory_states_memory_read_TCC1

Terse proof for `plain_memory_states_memory_read_TCC1`.
`plain_memory_states_memory_read_TCC1:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, s: State):
       plain_memory?(pm) ∧ pm`states(s) ∧ union(pm`ro_addr, pm`rw_addr)(addr) ⊃
         OK?[State, Byte](memory_read(pm`mem)(addr)(s)) ∨
           Exception?[State, Byte](memory_read(pm`mem)(addr)(s))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of `plain_memory_states_memory_read_TCC1`.
Q.E.D.

C.183.41 Plain_Memory.plain_memory_states_memory_read

Terse proof for `plain_memory_states_memory_read`.
`plain_memory_states_memory_read:`

```
{1}  ∀ (addr: Address, pm: Plain_Memory, s: State):
       plain_memory?(pm) ∧ pm`states(s) ∧ union(pm`ro_addr, pm`rw_addr)(addr) ⊃
         pm`states(state(memory_read(pm`mem)(addr)(s)))
```

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_invariant_read_transformers_ro_rw_addr,
Simplifying, rewriting, and recording with decision procedures,
Using lemma expr_transformer_invariant_next_ok[State, Byte],
Installing automatic rewrites from: memory_read_transformers_memory_read plain_memory_memory_read_ok has_next_state!
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_states_memory_read`.
Q.E.D.

C.183.42 Plain_Memory.plain_memory_memory_read_memory_read_TCC1

Terse proof for `plain_memory_memory_read_memory_read_TCC1`.

plain_memory_memory_read_memory_read_TCC1:

```
{1}  ∀ (pm: Plain_Memory, s: State, addr1, addr2: Address):
       plain_memory?(pm) ∧
        pm‘states(s) ∧
         union(pm‘ro_addr, pm‘rw_addr)(addr1) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr2)
       ⊃
       OK?[State, Byte](memory_read(pm‘mem)(addr2)(s)) ∨
        Exception?[State, Byte](memory_read(pm‘mem)(addr2)(s))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of plain_memory_memory_read_memory_read_TCC1.
Q.E.D.

C.183.43 Plain_Memory.plain_memory_memory_read_memory_read_TCC2

Terse proof for plain_memory_memory_read_memory_read_TCC2.

plain_memory_memory_read_memory_read_TCC2:

```
{1}  ∀ (pm: Plain_Memory, s: State, addr1, addr2: Address):
       plain_memory?(pm) ∧
        pm‘states(s) ∧
         union(pm‘ro_addr, pm‘rw_addr)(addr1) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr2)
       ⊃
       OK?[State, Byte]
            (memory_read(pm‘mem)
                  (addr1)(state[State, Byte](memory_read(pm‘mem)(addr2)(s))))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
Rewriting using plain_memory_states_memory_read, matching in *,
This completes the proof of plain_memory_memory_read_memory_read_TCC2.
Q.E.D.

C.183.44 Plain_Memory.plain_memory_memory_read_memory_read_TCC3

Terse proof for plain_memory_memory_read_memory_read_TCC3.

plain_memory_memory_read_memory_read_TCC3:

```
{1}  ∀ (pm: Plain_Memory, s: State, addr1, addr2: Address):
       plain_memory?(pm) ∧
        pm‘states(s) ∧
         union(pm‘ro_addr, pm‘rw_addr)(addr1) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr2)
       ⊃ OK?[State, Byte](memory_read(pm‘mem)(addr1)(s))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of plain_memory_memory_read_memory_read_TCC3.

Q.E.D.

C.183.45 Plain_Memory.plain_memory_memory_read_memory_read

Terse proof for `plain_memory_memory_read_memory_read`.

`plain_memory_memory_read_memory_read`:

{1} ∀ (pm: Plain_Memory, s: State, addr1, addr2: Address):
plain_memory?(pm) ∧
pm‘states(s) ∧
union(pm‘ro_addr, pm‘rw_addr)(addr1) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr2)
⊃
data(memory_read(pm‘mem)(addr1)(state(memory_read(pm‘mem)(addr2)(s)))) =
data(memory_read(pm‘mem)(addr1)(s))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_unchanged_read_ro_rw_addr,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_unchanged_memory_invariant_unchanged[State, Byte],

Installing automatic rewrites from: plain_memory_memory_read_ok plain_memory_states_memory_read memory_read_transformers_memory_read

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `plain_memory_memory_read_memory_read`.

Q.E.D.

C.183.46 Plain_Memory.plain_memory_memory_write_ok

Terse proof for `plain_memory_memory_write_ok`.

`plain_memory_memory_write_ok`:

{1} ∀ (addr: Address, pm: Plain_Memory, s: State, byte: Byte):
plain_memory?(pm) ∧ pm‘states(s) ∧ pm‘rw_addr(addr) ⊃
OK?(memory_write(pm‘mem)(addr, byte)(s))

Repeatedly Skolemizing and flattening,

Using lemma plain_memory_transformers_ok_write_rw,

Simplifying, rewriting, and recording with decision procedures,

Using lemma expr_transformers_ok_ok,

Installing automatic rewrites from: memory_write_transformers_memory_write

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `plain_memory_memory_write_ok`.

Q.E.D.

C.183.47 Plain_Memory.plain_memory_states_memory_write_TCC1

Terse proof for `plain_memory_states_memory_write_TCC1`.

plain_memory_states_memory_write_TCC1:

{1} ∀ (addr: Address, pm: Plain_Memory, s: State, byte: Byte):
plain_memory?(pm) ∧ pm‘states(s) ∧ pm‘rw_addr(addr) ⊃
OK?[State, Unit](memory_write(pm‘mem)(addr, byte)(s)) ∨
Exception?[State, Unit](memory_write(pm‘mem)(addr, byte)(s))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_write_ok, matching in *,
This completes the proof of plain_memory_states_memory_write_TCC1.
Q.E.D.

C.183.48 Plain_Memory.plain_memory_states_memory_write

Terse proof for plain_memory_states_memory_write.

plain_memory_states_memory_write:

{1} ∀ (addr: Address, pm: Plain_Memory, s: State, byte: Byte):
plain_memory?(pm) ∧ pm‘states(s) ∧ pm‘rw_addr(addr) ⊃
pm‘states(state(memory_write(pm‘mem)(addr, byte)(s)))

Repeatedly Skolemizing and flattening,
Using lemma plain_memory_invariant_write_transformers_rw_addr,
Using lemma expr_transformer_invariant_next_ok[State, Unit],
Installing automatic rewrites from: memory_write_transformers_memory_write plain_memory_memory_write_ok has_next_state!
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of plain_memory_states_memory_write.
Q.E.D.

C.183.49 Plain_Memory.plain_memory_memory_read_memory_write_other_TCC1

Terse proof for plain_memory_memory_read_memory_write_other_TCC1.

plain_memory_memory_read_memory_write_other_TCC1:

{1} ∀ (pm: Plain_Memory, s: State, ro_addr, rw_addr: Address, byte: Byte):
plain_memory?(pm) ∧
pm‘states(s) ∧
union(pm‘ro_addr, pm‘rw_addr)(ro_addr) ∧ pm‘rw_addr(rw_addr) ∧ ¬ ro_addr = rw_add
⊃
OK?[State, Unit](memory_write(pm‘mem)(rw_addr, byte)(s)) ∨
Exception?[State, Unit](memory_write(pm‘mem)(rw_addr, byte)(s))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_write_ok, matching in *,
This completes the proof of plain_memory_memory_read_memory_write_other_TCC1.
Q.E.D.

C.183.50 Plain_Memory.plain_memory_memory_read_memory_write_other_TCC2

Terse proof for `plain_memory_memory_read_memory_write_other_TCC2`.

`plain_memory_memory_read_memory_write_other_TCC2`:

{1} ∀ (pm: Plain_Memory, s: State, ro_addr, rw_addr: Address, byte: Byte):
plain_memory?(pm) ∧
pm‘states(s) ∧
union(pm‘ro_addr, pm‘rw_addr)(ro_addr) ∧ pm‘rw_addr(rw_addr) ∧ ¬ ro_addr = rw_addr
⊃
OK?[State, Byte]
(memory_read(pm‘mem)
(ro_addr)(state[State, Unit](memory_write(pm‘mem)(rw_addr, byte)(s))))

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_read_ok, matching in *,

Rewriting using plain_memory_states_memory_write, matching in *,

This completes the proof of `plain_memory_memory_read_memory_write_other_TCC2`.

Q.E.D.

C.183.51 Plain_Memory.plain_memory_memory_read_memory_write_other_TCC3

Terse proof for `plain_memory_memory_read_memory_write_other_TCC3`.

`plain_memory_memory_read_memory_write_other_TCC3`:

{1} ∀ (pm: Plain_Memory, s: State, ro_addr, rw_addr: Address):
plain_memory?(pm) ∧
pm‘states(s) ∧
union(pm‘ro_addr, pm‘rw_addr)(ro_addr) ∧ pm‘rw_addr(rw_addr) ∧ ¬ ro_addr = rw_addr
⊃ OK?[State, Byte](memory_read(pm‘mem)(ro_addr)(s))

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_memory_read_ok, matching in *,

This completes the proof of `plain_memory_memory_read_memory_write_other_TCC3`.

Q.E.D.

C.183.52 Plain_Memory.plain_memory_memory_read_memory_write_other

Terse proof for `plain_memory_memory_read_memory_write_other`.

`plain_memory_memory_read_memory_write_other:`

{1} ∀ (pm: Plain_Memory, s: State, ro_addr, rw_addr: Address, byte: Byte):
plain_memory?(pm) ∧
pm‘states(s) ∧
union(pm‘ro_addr, pm‘rw_addr)(ro_addr) ∧ pm‘rw_addr(rw_addr) ∧ ¬ ro_addr = rw_add
⊃
data(memory_read(pm‘mem)
(ro_addr)(state(memory_write(pm‘mem)(rw_addr, byte)(s))))
= data(memory_read(pm‘mem)(ro_addr)(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: union member
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`plain_memory_memory_read_memory_write_other.1:`

{-1} byte′ < max_byte
{-2} plain_memory?(pm′)
{-3} pm′‘states(s′)
{-4} pm′‘ro_addr(ro_addr′)
{-5} pm′‘rw_addr(rw_addr′)

{1} ro_addr′ = rw_addr′
{2} data(memory_read(pm′‘mem)
(ro_addr′)(state(memory_write(pm′‘mem)(rw_addr′, byte′)(s′))))
= data(memory_read(pm′‘mem)(ro_addr′)(s′))

Using lemma plain_memory_unchanged_memory_invariant_write,
Simplifying, rewriting, and recording with decision procedures,
Using lemma expr_unchanged_memory_invariant_unchanged[State, Unit],
Installing automatic rewrites from: plain_memory_memory_read_ok plain_memory_states_memory_write plain_memory_memory_write_ok memory_write_transformers_memory_write
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `plain_memory_memory_read_memory_write_other.1`.
`plain_memory_memory_read_memory_write_other.2:`

{-1} byte′ < max_byte
{-2} plain_memory?(pm′)
{-3} pm′‘states(s′)
{-4} pm′‘rw_addr(ro_addr′)
{-5} pm′‘rw_addr(rw_addr′)

{1} ro_addr′ = rw_addr′
{2} data(memory_read(pm′‘mem)
(ro_addr′)(state(memory_write(pm′‘mem)(rw_addr′, byte′)(s′))))
= data(memory_read(pm′‘mem)(ro_addr′)(s′))

Using lemma plain_memory_unchanged_memory_write_invariant,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of unchanged_memory_write_invariant?,
Instantiating the top quantifier in -1 with the terms: rw_addr′, byte′,
Simplifying, rewriting, and recording with decision procedures,
Using lemma expr_unchanged_memory_invariant_unchanged[State, Unit],
Installing automatic rewrites from: plain_memory_memory_read_ok plain_memory_states_memory_write plain_memory_memory_write_ok memory_write_transformers_memory_write singleton

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_memory_memory_read_memory_write_other.2`.
Q.E.D.

C.183.53 Plain_Memory.stays_unchanged_TCC1

Terse proof for `stays_unchanged_TCC1`.
`stays_unchanged_TCC1`:

{1} ∀ (pm: (plain_memory?), s_1, s_2: (pm‘states), addresses: PRED[Address], addr: Address):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧ addresses(addr) ⊃
OK?[State, Byte](memory_read(pm‘mem)(addr)(s_1))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of `stays_unchanged_TCC1`.
Q.E.D.

C.183.54 Plain_Memory.stays_unchanged_TCC2

Terse proof for `stays_unchanged_TCC2`.
`stays_unchanged_TCC2`:

{1} ∀ (pm: (plain_memory?), s_1, s_2: (pm‘states), addresses: PRED[Address], addr: Address):
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧ addresses(addr) ⊃
OK?[State, Byte](memory_read(pm‘mem)(addr)(s_2))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of `stays_unchanged_TCC2`.
Q.E.D.

C.183.55 Plain_Memory.stays_unchanged_unchanged_TCC1

Terse proof for `stays_unchanged_unchanged_TCC1`.
`stays_unchanged_unchanged_TCC1`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address], addr: Address):
plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
stays_unchanged(pm)(s_1, s_2, addresses) ∧ addresses(addr)
⊃ OK?[State, Byte](memory_read(pm‘mem)(addr)(s_1))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,

This completes the proof of `stays_unchanged_unchanged_TCC1`.
Q.E.D.

C.183.56 Plain_Memory.stays_unchanged_unchanged_TCC2

Terse proof for `stays_unchanged_unchanged_TCC2`.
`stays_unchanged_unchanged_TCC2`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address], addr: Address):
plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
stays_unchanged(pm)(s_1, s_2, addresses) ∧ addresses(addr)
⊃ OK?[State, Byte](memory_read(pm‘mem)(addr)(s_2))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
This completes the proof of `stays_unchanged_unchanged_TCC2`.
Q.E.D.

C.183.57 Plain_Memory.stays_unchanged_unchanged

Terse proof for `stays_unchanged_unchanged`.
`stays_unchanged_unchanged`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address], addr: Address):
plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
union(pm‘ro_addr, pm‘rw_addr)(addr) ∧
stays_unchanged(pm)(s_1, s_2, addresses) ∧ addresses(addr)
⊃ data(memory_read(pm‘mem)(addr)(s_1)) = data(memory_read(pm‘mem)(addr)(s_2))

Repeatedly Skolemizing and flattening,
Expanding the definition of stays_unchanged,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `stays_unchanged_unchanged`.
Q.E.D.

C.183.58 Plain_Memory.stays_unchanged_symmetric

Terse proof for `stays_unchanged_symmetric`.
`stays_unchanged_symmetric`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address]):
plain_memory?(pm) ∧ pm‘states(s_1) ∧ pm‘states(s_2) ⊃
stays_unchanged(pm)(s_1, s_2, addresses) = stays_unchanged(pm)(s_2, s_1, addresses)

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

we get 2 subgoals:
`stays_unchanged_symmetric.1:`

{-1} plain_memory?(pm′)
{-2} pm′‘states(s_1')
{-3} pm′‘states(s_2')
{-4} stays_unchanged(pm′)(s_1', s_2', addresses′)

{1} stays_unchanged(pm′)(s_2', s_1', addresses′)

Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `stays_unchanged_symmetric.1`.
`stays_unchanged_symmetric.2:`

{-1} plain_memory?(pm′)
{-2} pm′‘states(s_1')
{-3} pm′‘states(s_2')
{-4} stays_unchanged(pm′)(s_2', s_1', addresses′)

{1} stays_unchanged(pm′)(s_1', s_2', addresses′)

Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `stays_unchanged_symmetric.2`.
Q.E.D.

C.183.59 Plain_Memory.stays_unchanged_mono

Terse proof for `stays_unchanged_mono`.
`stays_unchanged_mono:`

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses_1, addresses_2: PRED⌊Address⌋):
plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
(addresses_1 ⊆ addresses_2) ∧ stays_unchanged(pm)(s_1, s_2, addresses_2)
⊃ stays_unchanged(pm)(s_1, s_2, addresses_1)

Repeatedly Skolemizing and flattening,
Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Hiding formulas: -1, 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stays_unchanged_mono`.
Q.E.D.

C.183.60 Plain_Memory.stays_unchanged_only_changes

Terse proof for `stays_unchanged_only_changes`.

`stays_unchanged_only_changes:`

{1}	∀ (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address]): plain_memory?(pm) ∧ pm‘states(s_1) ∧ pm‘states(s_2) ∧ (pm‘ro_addr ⊆ addresses) ⊃ stays_unchanged(pm)(s_1, s_2, addresses) = only_changes(pm)(s_1, s_2, (pm‘rw_addr \ addresses))

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`stays_unchanged_only_changes.1:`

{-1}	plain_memory?(pm′)
{-2}	pm′‘states(s_1')
{-3}	pm′‘states(s_2')
{-4}	(pm′‘ro_addr ⊆ addresses′)
{-5}	stays_unchanged(pm′)(s_1', s_2', addresses′)
{1}	only_changes(pm′)(s_1', s_2', (pm′‘rw_addr \ addresses′))

Expanding the definition of only_changes,
Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Keeping (-4 -5 -6 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stays_unchanged_only_changes.1`.
`stays_unchanged_only_changes.2:`

{-1}	plain_memory?(pm′)
{-2}	pm′‘states(s_1')
{-3}	pm′‘states(s_2')
{-4}	(pm′‘ro_addr ⊆ addresses′)
{-5}	only_changes(pm′)(s_1', s_2', (pm′‘rw_addr \ addresses′))
{1}	stays_unchanged(pm′)(s_1', s_2', addresses′)

Expanding the definition of only_changes,
Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Keeping (-4 -5 -6 2) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stays_unchanged_only_changes.2`.
Q.E.D.

C.183.61 Plain_Memory.stays_unchanged_only_changes_disjoint

Terse proof for `stays_unchanged_only_changes_disjoint`.

`stays_unchanged_only_changes_disjoint`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses_1, addresses_2: PRED[Address]):
plain_memory?(pm) ∧
pm`states(s_1) ∧
pm`states(s_2) ∧
(addresses_2 ⊆ (pm`ro_addr ∪ pm`rw_addr)) ∧
disjoint?(addresses_1, addresses_2) ∧ only_changes(pm)(s_1, s_2, addresses_1)
⊃ stays_unchanged(pm)(s_1, s_2, addresses_2)

Repeatedly Skolemizing and flattening,
Expanding the definition of only_changes,
Using lemma stays_unchanged_mono,
Simplifying, rewriting, and recording with decision procedures,
Keeping (-4 -5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stays_unchanged_only_changes_disjoint`.
Q.E.D.

C.183.62 Plain_Memory.only_changes_symmetric

Terse proof for `only_changes_symmetric`.

`only_changes_symmetric`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address]):
plain_memory?(pm) ∧ pm`states($s_1$) ∧ pm`states(s_2) ⊃
only_changes(pm)(s_1, s_2, addresses) = only_changes(pm)(s_2, s_1, addresses)

Repeatedly Skolemizing and flattening,
Expanding the definition of only_changes,
Rewriting using stays_unchanged_symmetric, matching in *,
This completes the proof of `only_changes_symmetric`.
Q.E.D.

C.183.63 Plain_Memory.only_changes_mono

Terse proof for `only_changes_mono`.

`only_changes_mono`:

{1} ∀ (pm: Plain_Memory, s_1, s_2: State, addresses_1, addresses_2: PRED[Address]):
plain_memory?(pm) ∧
pm`states(s_1) ∧
pm`states(s_2) ∧
(addresses_1 ⊆ addresses_2) ∧ only_changes(pm)(s_1, s_2, addresses_1)
⊃ only_changes(pm)(s_1, s_2, addresses_2)

Repeatedly Skolemizing and flattening,
Expanding the definition of only_changes,
Using lemma stays_unchanged_mono,
Simplifying, rewriting, and recording with decision procedures,
Keeping (-4 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `only_changes_mono`.
Q.E.D.

C.183.64 Plain_Memory.only_changes_unchanged_TCC1

Terse proof for `only_changes_unchanged_TCC1`.

`only_changes_unchanged_TCC1`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, s1, s2: State, addresses: PRED[Address]):
       plain_memory?(pm) ∧
        pm‘states(s1) ∧
         pm‘states(s2) ∧
          only_changes(pm)(s1, s2, addresses) ∧
           (pm‘ro_addr(addr) ∨ pm‘rw_addr(addr) ∧ ¬ addresses(addr))
        ⊃ OK?[State, Byte](memory_read(pm‘mem)(addr)(s1))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `only_changes_unchanged_TCC1`.
Q.E.D.

C.183.65 Plain_Memory.only_changes_unchanged_TCC2

Terse proof for `only_changes_unchanged_TCC2`.

`only_changes_unchanged_TCC2`:

```
{1}  ∀ (addr: Address, pm: Plain_Memory, s1, s2: State, addresses: PRED[Address]):
       plain_memory?(pm) ∧
        pm‘states(s1) ∧
         pm‘states(s2) ∧
          only_changes(pm)(s1, s2, addresses) ∧
           (pm‘ro_addr(addr) ∨ pm‘rw_addr(addr) ∧ ¬ addresses(addr))
        ⊃ OK?[State, Byte](memory_read(pm‘mem)(addr)(s2))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_ok, matching in *,
Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `only_changes_unchanged_TCC2`.
Q.E.D.

C.183.66 Plain_Memory.only_changes_unchanged

Terse proof for `only_changes_unchanged`.

only_changes_unchanged:

{1} ∀ (addr: Address, pm: Plain_Memory, s_1, s_2: State, addresses: PRED[Address]):
plain_memory?(pm) ∧
pm‘states(s_1) ∧
pm‘states(s_2) ∧
only_changes(pm)(s_1, s_2, addresses) ∧
(pm‘ro_addr(addr) ∨ pm‘rw_addr(addr) ∧ ¬ addresses(addr))
⊃ data(memory_read(pm‘mem)(addr)(s_1)) = data(memory_read(pm‘mem)(addr)(s_2))

Repeatedly Skolemizing and flattening,
Expanding the definition of only_changes,
Using lemma stays_unchanged_unchanged,
Simplifying, rewriting, and recording with decision procedures,
Keeping (-5 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of only_changes_unchanged.
Q.E.D.

C.183.67 Plain_Memory.plain_memory_stays_unchanged_memory_read_TCC1

Terse proof for plain_memory_stays_unchanged_memory_read_TCC1.
plain_memory_stays_unchanged_memory_read_TCC1:

{1} ∀ (pm: Plain_Memory, s: State, addr: Address):
plain_memory?(pm) ∧ pm‘states(s) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr) ⊃
pm‘states(state[State, Byte](memory_read(pm‘mem)(addr)(s)))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_states_memory_read, matching in *,
This completes the proof of plain_memory_stays_unchanged_memory_read_TCC1.
Q.E.D.

C.183.68 Plain_Memory.plain_memory_stays_unchanged_memory_read

Terse proof for plain_memory_stays_unchanged_memory_read.
plain_memory_stays_unchanged_memory_read:

{1} ∀ (pm: Plain_Memory, s: State, addr: Address):
plain_memory?(pm) ∧ pm‘states(s) ∧ union(pm‘ro_addr, pm‘rw_addr)(addr) ⊃
stays_unchanged(pm)
(s, state(memory_read(pm‘mem)(addr)(s)),
(pm‘ro_addr ∪ pm‘rw_addr))

Repeatedly Skolemizing and flattening,
Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_memory_read, matching in *,
This completes the proof of plain_memory_stays_unchanged_memory_read.

Q.E.D.

C.183.69 Plain_Memory.plain_memory_stays_unchanged_memory_write_TCC1

Terse proof for `plain_memory_stays_unchanged_memory_write_TCC1`.
`plain_memory_stays_unchanged_memory_write_TCC1:`

```
{1}  ∀ (pm: Plain_Memory, s: State, addr: Address, byte: Byte):
       plain_memory?(pm) ∧ pm‘states(s) ∧ pm‘rw_addr(addr) ⊃
         pm‘states(state[State, Unit](memory_write(pm‘mem)(addr, byte)(s)))
```

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_states_memory_write, matching in *,
This completes the proof of `plain_memory_stays_unchanged_memory_write_TCC1`.
Q.E.D.

C.183.70 Plain_Memory.plain_memory_stays_unchanged_memory_write

Terse proof for `plain_memory_stays_unchanged_memory_write`.
`plain_memory_stays_unchanged_memory_write:`

```
{1}  ∀ (pm: Plain_Memory, s: State, addr: Address, byte: Byte):
       plain_memory?(pm) ∧ pm‘states(s) ∧ pm‘rw_addr(addr) ⊃
         stays_unchanged(pm)
                        (s, state(memory_write(pm‘mem)(addr, byte)(s)),
                          ((pm‘ro_addr ∪ pm‘rw_addr) \ {addr}))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of stays_unchanged,
Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_memory_read_memory_write_other, matching in *,
Expanding the definition of remove,
which is trivially true.
This completes the proof of `plain_memory_stays_unchanged_memory_write`.
Q.E.D.

C.184 Proofs for PointerToMemberExpressions (expressions.pvs)

C.184.1 PointerToMemberExpressions.ptm_member_TCC1

Terse proof for `ptm_member_TCC1`.
`ptm_member_TCC1:`

```
{1}  ∀ (typ: Cpp_Subtype(cv(pointer_to_member?))): pointer_to_member?(cv_base(typ))
```

Repeatedly Skolemizing and flattening,

Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ptm_member_TCC1`.
Q.E.D.

C.184.2 PointerToMemberExpressions.ptm_arrow_TCC1

Terse proof for `ptm_arrow_TCC1`.

`ptm_arrow_TCC1:`

```
{1}  ∀ ((ptm_typ: Cpp_Subtype(cv(pointer_to_member?))),
          ofs: (range_ptm(cv_base(ptm_typ)))):
        Cpp_Type?(pointer(typ(cv_base(ptm_typ)))) ∧
         cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                   (λ (t: Cpp_Type):
                        pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t)))))
           (pointer(typ(cv_base(ptm_typ))))
```

Repeatedly Skolemizing and flattening,
Using lemma cv_base_result,
we get 2 subgoals:

`ptm_arrow_TCC1.1:`

```
{-1}  pointer_to_member?(cv_base(ptm_typ′))
{-2}  Cpp_Type?(ptm_typ′)
{-3}  cv(pointer_to_member?)(ptm_typ′)
{-4}  range_ptm(cv_base(ptm_typ′))(ofs′)
{1}   Cpp_Type?(pointer(typ(cv_base(ptm_typ′)))) ∧
       cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                 (λ (t: Cpp_Type): pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t)))))
         (pointer(typ(cv_base(ptm_typ′))))
```

Case splitting on Cpp_Type?(pointer(typ(cv_base(ptm_typ!1)))),
we get 3 subgoals:

`ptm_arrow_TCC1.1.1:`

```
{-1}  Cpp_Type?(pointer(typ(cv_base(ptm_typ′))))
{-2}  pointer_to_member?(cv_base(ptm_typ′))
{-3}  Cpp_Type?(ptm_typ′)
{-4}  cv(pointer_to_member?)(ptm_typ′)
{-5}  range_ptm(cv_base(ptm_typ′))(ofs′)
{1}   Cpp_Type?(pointer(typ(cv_base(ptm_typ′)))) ∧
       cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
                 (λ (t: Cpp_Type): pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t)))))
         (pointer(typ(cv_base(ptm_typ′))))
```

Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of extend,
Keeping (-1 -2 -4 1) and hiding *,
Adding type constraints for typ(cv_base(ptm_typ!1)),
Simplifying, rewriting, and recording with decision procedures,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ptm_arrow_TCC1.1.1`.

`ptm_arrow_TCC1.1.2:`

{-1} pointer_to_member?(cv_base(ptm_typ′))
{-2} Cpp_Type?(ptm_typ′)
{-3} cv(pointer_to_member?)(ptm_typ′)
{-4} range_ptm(cv_base(ptm_typ′))(ofs′)

{1} Cpp_Type?(pointer(typ(cv_base(ptm_typ′))))
{2} Cpp_Type?(pointer(typ(cv_base(ptm_typ′)))) ∧
cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
(λ (t: Cpp_Type): pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t)))))
(pointer(typ(cv_base(ptm_typ′))))

Hiding formulas: 2,

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Splitting conjunctions,

we get 2 subgoals:

ptm_arrow_TCC1.1.2.1:

{-1} $t' = \text{pointer}(\text{typ}(\text{cv\_base}(\text{ptm\_typ}')))$
{-2} pointer_to_member?(cv_base(ptm_typ′))
{-3} $\forall$ (t: Cpp_Type_):
subterm(t, ptm_typ′) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)
{-4} cv(pointer_to_member?)(ptm_typ′)
{-5} range_ptm(cv_base(ptm_typ′))(ofs′)

{1} no_pointers_to_bitfield?(t') $\wedge$
no_pointers_to_references?(t') $\wedge$
no_cv_references?(t') $\wedge$
no_reference_to_reference?(t') $\wedge$
no_reference_to_bitfields?(t') $\wedge$
no_pointer_to_member_to_reference?(t') $\wedge$
no_pointer_to_member_to_cv_void?(t') $\wedge$
no_cv_void_parameter?(t') $\wedge$
no_array_of_references?(t') $\wedge$
no_array_of_cv_void?(t') $\wedge$
no_array_of_function?(t') $\wedge$
no_array_of_abstract_class?(t') $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t') $\wedge$
no_array_return_type?(t') $\wedge$
no_function_return_type?(t') $\wedge$
bitfield_underlying_integral_or_enum_type?(t') $\wedge$
cv_array?(t') $\wedge$
enum_underlying_integral?(t') $\wedge$
enum_constants?(t') $\wedge$
const_volatile?(t') $\wedge$
const_stutter?(t') $\wedge$
volatile_stutter?(t') $\wedge$
no_cv_class?(t') $\wedge$
no_cv_union?(t') $\wedge$
no_cv_function?(t') $\wedge$
no_reference_to_void?(t') $\wedge$
no_cv_void?(t') $\wedge$ no_array_of_bitfields?(t')

Hiding formulas: -3,

Replacing using formula -1,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_par no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?

Adding type constraints for typ(cv_base(ptm_typ!1)),

Simplifying, rewriting, and recording with decision procedures,

Keeping (-1 -3) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `ptm_arrow_TCC1.1.2.1`.

ptm_arrow_TCC1.1.2.2:

{-1} subterm(t', typ(cv_base(ptm_typ$'$)))
{-2} pointer_to_member?(cv_base(ptm_typ$'$))
{-3} $\forall$ (t: Cpp_Type_):
subterm(t, ptm_typ$'$) $\supset$
no_pointers_to_bitfield?(t) $\wedge$
no_pointers_to_references?(t) $\wedge$
no_cv_references?(t) $\wedge$
no_reference_to_reference?(t) $\wedge$
no_reference_to_bitfields?(t) $\wedge$
no_pointer_to_member_to_reference?(t) $\wedge$
no_pointer_to_member_to_cv_void?(t) $\wedge$
no_cv_void_parameter?(t) $\wedge$
no_array_of_references?(t) $\wedge$
no_array_of_cv_void?(t) $\wedge$
no_array_of_function?(t) $\wedge$
no_array_of_abstract_class?(t) $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t) $\wedge$
no_array_return_type?(t) $\wedge$
no_function_return_type?(t) $\wedge$
bitfield_underlying_integral_or_enum_type?(t) $\wedge$
cv_array?(t) $\wedge$
enum_underlying_integral?(t) $\wedge$
enum_constants?(t) $\wedge$
const_volatile?(t) $\wedge$
const_stutter?(t) $\wedge$
volatile_stutter?(t) $\wedge$
no_cv_class?(t) $\wedge$
no_cv_union?(t) $\wedge$
no_cv_function?(t) $\wedge$
no_reference_to_void?(t) $\wedge$
no_cv_void?(t) $\wedge$ no_array_of_bitfields?(t)
{-4} cv(pointer_to_member?)(ptm_typ$'$)
{-5} range_ptm(cv_base(ptm_typ$'$))(ofs$'$)

{1} no_pointers_to_bitfield?(t') $\wedge$
no_pointers_to_references?(t') $\wedge$
no_cv_references?(t') $\wedge$
no_reference_to_reference?(t') $\wedge$
no_reference_to_bitfields?(t') $\wedge$
no_pointer_to_member_to_reference?(t') $\wedge$
no_pointer_to_member_to_cv_void?(t') $\wedge$
no_cv_void_parameter?(t') $\wedge$
no_array_of_references?(t') $\wedge$
no_array_of_cv_void?(t') $\wedge$
no_array_of_function?(t') $\wedge$
no_array_of_abstract_class?(t') $\wedge$
no_pointer_or_ref_to_incomplete_array_parameter?(t') $\wedge$
no_array_return_type?(t') $\wedge$
no_function_return_type?(t') $\wedge$
bitfield_underlying_integral_or_enum_type?(t') $\wedge$
cv_array?(t') $\wedge$
enum_underlying_integral?(t') $\wedge$
enum_constants?(t') $\wedge$
const_volatile?(t') $\wedge$
const_stutter?(t') $\wedge$
volatile_stutter?(t') $\wedge$
no_cv_class?(t') $\wedge$
no_cv_union?(t') $\wedge$
no_cv_function?(t') $\wedge$
no_reference_to_void?(t') $\wedge$
no_cv_void?(t') $\wedge$ no_array_of_bitfields?(t')

Instantiating the top quantifier in -3 with the terms: (t!1),

Splitting conjunctions,

we get 2 subgoals:

ptm_arrow_TCC1.1.2.2.1:

{-1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

{-2} $\text{subterm}(t', \text{typ}(\text{cv\_base}(\text{ptm\_typ}')))$

{-3} $\text{pointer\_to\_member?}(\text{cv\_base}(\text{ptm\_typ}'))$

{-4} $\text{cv}(\text{pointer\_to\_member?})(\text{ptm\_typ}')$

{-5} $\text{range\_ptm}(\text{cv\_base}(\text{ptm\_typ}'))(\text{ofs}')$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

which is trivially true.

This completes the proof of `ptm_arrow_TCC1.1.2.2.1`.

`ptm_arrow_TCC1.1.2.2.2`:

{-1} subterm(t', typ(cv_base(ptm_typ$'$)))
{-2} pointer_to_member?(cv_base(ptm_typ$'$))
{-3} cv(pointer_to_member?)(ptm_typ$'$)
{-4} range_ptm(cv_base(ptm_typ$'$))(ofs$'$)

{1} subterm(t', ptm_typ$'$)
{2} no_pointers_to_bitfield?(t') ∧
no_pointers_to_references?(t') ∧
no_cv_references?(t') ∧
no_reference_to_reference?(t') ∧
no_reference_to_bitfields?(t') ∧
no_pointer_to_member_to_reference?(t') ∧
no_pointer_to_member_to_cv_void?(t') ∧
no_cv_void_parameter?(t') ∧
no_array_of_references?(t') ∧
no_array_of_cv_void?(t') ∧
no_array_of_function?(t') ∧
no_array_of_abstract_class?(t') ∧
no_pointer_or_ref_to_incomplete_array_parameter?(t') ∧
no_array_return_type?(t') ∧
no_function_return_type?(t') ∧
bitfield_underlying_integral_or_enum_type?(t') ∧
cv_array?(t') ∧
enum_underlying_integral?(t') ∧
enum_constants?(t') ∧
const_volatile?(t') ∧
const_stutter?(t') ∧
volatile_stutter?(t') ∧
no_cv_class?(t') ∧
no_cv_union?(t') ∧
no_cv_function?(t') ∧
no_reference_to_void?(t') ∧
no_cv_void?(t') ∧ no_array_of_bitfields?(t')

Hiding formulas: 2,

Using lemma subterm_cv_base,

Adding type constraints for typ(cv_base(ptm_typ!1)),

Simplifying, rewriting, and recording with decision procedures,

Expanding the definition of subterm,

Replacing using formula -3,

Using lemma subterm_transitive,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of subterm,

which is trivially true.

This completes the proof of `ptm_arrow_TCC1.1.2.2.2`.

`ptm_arrow_TCC1.1.3`:

{-1}	pointer_to_member?(cv_base(ptm_typ′))
{-2}	Cpp_Type?(ptm_typ′)
{-3}	cv(pointer_to_member?)(ptm_typ′)
{-4}	range_ptm(cv_base(ptm_typ′))(ofs′)
{1}	array?(cv_base(ptm_typ′)) ∨ pointer?(cv_base(ptm_typ′)) ∨ reference?(cv_base(ptm_typ′)) ∨ bitfield?(cv_base(ptm_typ′)) ∨ enum?(cv_base(ptm_typ′)) ∨ pointer_to_member?(cv_base(ptm_typ′)) ∨ const?(cv_base(ptm_typ′)) ∨ volatile?(cv_base(ptm_typ′))
{2}	Cpp_Type?(pointer(typ(cv_base(ptm_typ′)))) ∧ cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] (λ (t: Cpp_Type): pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t))))) (pointer(typ(cv_base(ptm_typ′))))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `ptm_arrow_TCC1.1.3`.

`ptm_arrow_TCC1.2`:

{-1}	Cpp_Type?(ptm_typ′)
{-2}	cv(pointer_to_member?)(ptm_typ′)
{-3}	range_ptm(cv_base(ptm_typ′))(ofs′)
{1}	(pointer_to_member? ⊆ interpreted?)
{2}	Cpp_Type?(pointer(typ(cv_base(ptm_typ′)))) ∧ cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE] (λ (t: Cpp_Type): pointer?(t) ∧ (class?(typ(t)) ∨ union?(typ(t))))) (pointer(typ(cv_base(ptm_typ′))))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ptm_arrow_TCC1.2`.
Q.E.D.

C.185 Proofs for Posnat_induction (vfiasco-prelude.pvs)

C.185.1 Posnat_induction.posnat_induction

Terse proof for `posnat_induction`.

`posnat_induction`:

{1}	∀ (p: pred[posnat]): (p(1) ∧ (∀ j: p(j) ⊃ p(j + 1))) ⊃ (∀ i: p(i))

For the top quantifier in 1, we introduce Skolem constants: (p′),
Applying disjunctive simplification to flatten sequent,
Inducting on i on formula 1,
we get 3 subgoals:

`posnat_induction.1:`

{-1}	$p'(1)$
{-2}	$\forall\ j\colon\ p'(j)\ \supset\ p'(j+1)$
{1}	$i' > 0$
{2}	$p'(i')$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `posnat_induction.1`.
`posnat_induction.2:`

{-1}	$p'(1)$
{-2}	$\forall\ j\colon\ p'(j)\ \supset\ p'(j+1)$
{1}	$0 > 0\ \supset\ p'(0)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `posnat_induction.2`.
`posnat_induction.3:`

{-1}	$p'(1)$
{-2}	$\forall\ j\colon\ p'(j)\ \supset\ p'(j+1)$
{1}	$\forall\ j\colon\ (j > 0\ \supset\ p'(j))\ \supset\ j+1 > 0\ \supset\ p'(j+1)$

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
we get 2 subgoals:
`posnat_induction.3.1:`

{-1}	$j' \geq 0$
{-2}	$j' > 0\ \supset\ p'(j')$
{-3}	$j'+1 > 0$
{-4}	$p'(1)$
{-5}	$p'(j')\ \supset\ p'(j'+1)$
{1}	$p'(j'+1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `posnat_induction.3.1`.
`posnat_induction.3.2:`

{-1}	$j' \geq 0$
{-2}	$j' > 0\ \supset\ p'(j')$
{-3}	$j'+1 > 0$
{-4}	$p'(1)$
{1}	$j' > 0$
{2}	$p'(j'+1)$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `posnat_induction.3.2`.
Q.E.D.

C.186 Proofs for PostfixExpressions (expressions.pvs)

C.186.1 PostfixExpressions.postinc_TCC1

Terse proof for `postinc_TCC1`.

postinc_TCC1:

{1} ∀ ((typ: Cpp_Subtype(v(non_bool_integral?)))): cv(non_bool_integral_enum?)(typ)

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of postinc_TCC1.
Q.E.D.

C.186.2 PostfixExpressions.postinc_float_TCC1

Terse proof for postinc_float_TCC1.
postinc_float_TCC1:

{1} ∀ ((typ: Cpp_Subtype(v(floating_point?)))): cv(floating_point?)(typ)

Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of postinc_float_TCC1.
Q.E.D.

C.186.3 PostfixExpressions.postinc_float_TCC2

Terse proof for postinc_float_TCC2.
postinc_float_TCC2:

{1} ∀ ((typ: Cpp_Subtype(v(floating_point?)))): floating_point?(cv_base(typ))

Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
we get 2 subgoals:
postinc_float_TCC2.1:

{-1}	Cpp_Type?(typ′)
{-2}	v(floating_point?)(typ′)
{1}	cv(floating_point?)(typ′)
{2}	floating_point?(cv_base(typ′))

Using lemma postinc_float_TCC1,
This completes the proof of postinc_float_TCC2.1.
postinc_float_TCC2.2:

{-1}	Cpp_Type?(typ′)
{-2}	v(floating_point?)(typ′)
{1}	(floating_point? ⊆ interpreted?)
{2}	floating_point?(cv_base(typ′))

Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of postinc_float_TCC2.2.
Q.E.D.

C.186.4 PostfixExpressions.postinc_float_TCC3

Terse proof for `postinc_float_TCC3`.
`postinc_float_TCC3:`

```
{1}  ∀ ((typ: Cpp_Subtype(v(floating_point?))),
          n: (range_floating_point(cv_base(typ)))):
       cv(floating_point?)(cv_base(typ))
```

Repeatedly Skolemizing and flattening,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `postinc_float_TCC3`.
Q.E.D.

C.187 Proofs for PrimaryExpressions (expressions.pvs)

This theory contains no provable formal statements.

C.188 Proofs for Pte_type (paging-data.pvs)

This theory contains no provable formal statements.

C.189 Proofs for Pte_type_adt (Pte_type_adt.pvs)

C.189.1 Pte_type_adt.Pte_TCC1

Terse proof for `Pte_TCC1`.
`Pte_TCC1:`

```
{1}  ∀ (a: Memory_Address_4G): offset(a) ≥ 0
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Pte_TCC1`.
Q.E.D.

C.190 Proofs for Pte_type_adt_reduce (Pte_type_adt.pvs)

This theory contains no provable formal statements.

C.191 Proofs for Random_Device (random_device.pvs)

C.191.1 Random_Device.base_TCC1

Terse proof for `base_TCC1`.

base_TCC1:

{1} $\exists$ (x: ({m: Memory_Address_4G | $m + \text{size} \leq \text{max\_linear}$})): TRUE

Instantiating the top quantifier in 1 with the terms: (Mem(0)),
we get 2 subgoals:
base_TCC1.1:

{1} TRUE

which is trivially true.
This completes the proof of base_TCC1.1.
base_TCC1.2:

{1} Mem?(Mem(0)`type_of) $\wedge$
(0 $\leq$ Mem(0)`offset $\wedge$ Mem(0)`offset $<$ max_linear_offset) $\wedge$
Mem(0) + size $\leq$ max_linear

Using lemma dt_uint_size,
Expanding the definition of max_linear,
Expanding the definition of Mem,
Expanding the definition of +,
Expanding the definition of size,
Expanding the definition of <=,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of base_TCC1.2.
Q.E.D.

C.191.2 Random_Device.rnd_seed_TCC1

Terse proof for rnd_seed_TCC1.
rnd_seed_TCC1:

{1} $0 \leq (\text{base} + 0 \times \text{size}(\text{uidt}(\text{dt\_uint})))\text{`offset} \wedge$
$(\text{base} + 0 \times \text{size}(\text{uidt}(\text{dt\_uint})))\text{`offset} < \text{max\_linear\_offset}$

Installing automatic rewrites from: + < ertimes_ax
Adding type constraints for base,
Expanding the definition of size,
Using lemma dt_uint_size,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of rnd_seed_TCC1.
Q.E.D.

C.191.3 Random_Device.mem_cnt_TCC1

Terse proof for mem_cnt_TCC1.
mem_cnt_TCC1:

{1} $0 \leq (\text{base} + 1 \times \text{size}(\text{uidt}(\text{dt\_uint})))\text{`offset} \wedge$
$(\text{base} + 1 \times \text{size}(\text{uidt}(\text{dt\_uint})))\text{`offset} < \text{max\_linear\_offset}$

Using lemma dt_uint_size,
Adding type constraints for base,
Installing automatic rewrites from: + <= < max_linear size Mem ertimes_ax
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `mem_cnt_TCC1`.
Q.E.D.

C.191.4 Random_Device.rnd_val_TCC1

Terse proof for `rnd_val_TCC1`.

`rnd_val_TCC1:`

{1} $0 \leq (\text{base} + 2 \times \text{size}(\text{uidt}(\text{dt\_uint})))\text{`offset} \wedge$
$(\text{base} + 2 \times \text{size}(\text{uidt}(\text{dt\_uint})))\text{`offset} < \text{max\_linear\_offset}$

Using lemma dt_uint_size,
Adding type constraints for base,
Installing automatic rewrites from: + <= < max_linear size Mem ertimes_ax
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `rnd_val_TCC1`.
Q.E.D.

C.191.5 Random_Device.increase_access_count_TCC1

Terse proof for `increase_access_count_TCC1`.

`increase_access_count_TCC1:`

{1} $\forall$ (s: Random_device_memory, n: nat):
$s\text{`expansion`counter\_offset} \leq n + \text{access\_count}(s)$

Repeatedly Skolemizing and flattening,
Expanding the definition of access_count,
Adding type constraints for s!1`expansion`counter_offset,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `increase_access_count_TCC1`.
Q.E.D.

C.191.6 Random_Device.clear_counter_TCC1

Terse proof for `clear_counter_TCC1`.

`clear_counter_TCC1:`

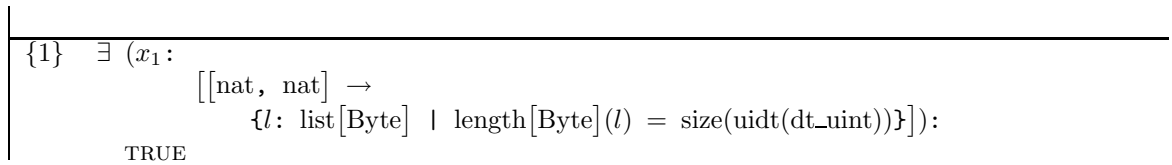
{1} $\forall$ (s: Random_device_memory): $\text{access\_count}(s) \leq s\text{`expansion`memory\_access\_count}$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `clear_counter_TCC1`.
Q.E.D.

C.191.7 Random_Device.random_TCC1

Terse proof for `random_TCC1`.

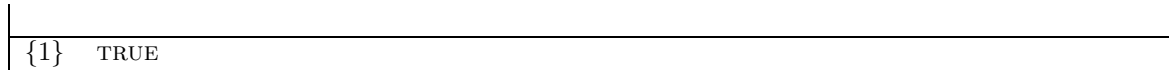
`random_TCC1:`



Instantiating the top quantifier in 1 with the terms: (λ (n1, n2: nat): choose(λ (l: list[Byte]): length(l) = size(uidt(dt_uint)))),

we get 2 subgoals:

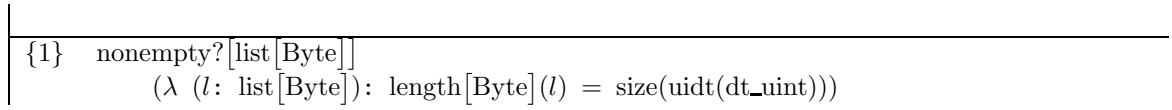
`random_TCC1.1:`



which is trivially true.

This completes the proof of `random_TCC1.1`.

`random_TCC1.2:`



Expanding the definition of nonempty?,

Expanding the definition of empty?,

Expanding the definition of member,

Rewriting using forall_not, matching in *,

Using lemma list_all_length[Byte],

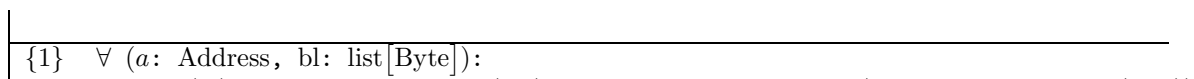
This completes the proof of `random_TCC1.2`.

Q.E.D.

C.191.8 Random_Device.read_mem_cnt_TCC1

Terse proof for `read_mem_cnt_TCC1`.

`read_mem_cnt_TCC1:`



Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `read_mem_cnt_TCC1`.

Q.E.D.

C.191.9 Random_Device.read_mem_cnt_TCC2

Terse proof for `read_mem_cnt_TCC2`.

`read_mem_cnt_TCC2:`

```
{1}  ∀ (a: Address, bl: list[Byte]):
       cons?(bl) ∧ a = mem_cnt ⊃
         (∀ (s: Random_device_memory): max_value(uint) ≥ 0 ∧ max_value(uint) > 0)
```

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Expanding the definition of max_value,

Expanding the definition of range_integral,

Using lemma max_uint,

Using lemma max_uint_bits,

Using lemma pos_expt_gt,

Case splitting on extend[real, int, bool, FALSE] (extend[int, nat, bool, FALSE](range_uint)) = extend[real, nat, bool, FALSE](range_uint),

we get 2 subgoals:

`read_mem_cnt_TCC2.1:`

```
{-1}  extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
        extend[real, nat, bool, FALSE](range_uint)
{-2}  max_value_bits_uint < 2 ^ max_value_bits_uint
{-3}  max_value_bits_uint ≥ 16
{-4}  max(range_uint) ≥ (2 ^ max_value_bits_uint) − 1
|-------
{1}   max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) ≥ 0 ∧
        max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) > 0
```

Replacing using formula -1,

Using lemma max_stable_under_extension[real, nat, ≤],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_mem_cnt_TCC2.1`.

`read_mem_cnt_TCC2.2:`

```
{-1}  max_value_bits_uint < 2 ^ max_value_bits_uint
{-2}  max_value_bits_uint ≥ 16
{-3}  max(range_uint) ≥ (2 ^ max_value_bits_uint) − 1
|-------
{1}   extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
        extend[real, nat, bool, FALSE](range_uint)
{2}   max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) ≥ 0 ∧
        max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) > 0
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `read_mem_cnt_TCC2.2`.

Q.E.D.

C.191.10 Random_Device.read_mem_cnt_TCC3

Terse proof for `read_mem_cnt_TCC3`.

`read_mem_cnt_TCC3:`

{1} $\forall$ (a: Address, bl: list[Byte]):
cons?(bl) $\wedge$ a = mem_cnt $\supset$
($\forall$ (s: Random_device_memory):
(range_uint)(rem(max_value(uint))
(access_count(s) − counter_offset(s))))

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Adding type constraints for rem(max_value(uint))(access_count(s!1) - counter_offset(s!1)),
we get 2 subgoals:
`read_mem_cnt_TCC3.1:`

{-1} rem(max_value(uint))(access_count(s') − counter_offset(s')) < max_value(uint)
{-2} $\exists$ q:
access_count(s') − counter_offset(s') =
rem(max_value(uint))(access_count(s') − counter_offset(s')) + max_value(uint) × q
---
{1} (range_uint)(rem(max_value(uint))(access_count(s') − counter_offset(s')))

Using lemma binary_range_uint,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Keeping (-1 2) and hiding *,
Expanding the definition of max_value,
Expanding the definition of range_integral,
Case splitting on extend[real, int, bool, FALSE] (extend[int, nat, bool, FALSE](range_uint)) = extend[real, nat, bool, FALSE](range_uint),
we get 2 subgoals:
`read_mem_cnt_TCC3.1.1:`

{-1} extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
extend[real, nat, bool, FALSE](range_uint)
{-2} rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
(access_count(s') − counter_offset(s'))
< max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)))

{1} rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
(access_count(s') − counter_offset(s'))
≤ max(range_uint)

Replacing using formula -1,
Using lemma max_stable_under_extension[real, nat, ≤],
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `read_mem_cnt_TCC3.1.1`.
`read_mem_cnt_TCC3.1.2:`

{-1} rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
(access_count(s') − counter_offset(s'))
< max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)))

{1} extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
extend[real, nat, bool, FALSE](range_uint)
{2} rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
(access_count(s') − counter_offset(s'))
≤ max(range_uint)

Keeping (1) and hiding *,

Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `read_mem_cnt_TCC3.1.2`.
`read_mem_cnt_TCC3.2`:

{1} max_value(uint) ≥ 0
{2} (range_uint)(rem(max_value(uint))(access_count(s') − counter_offset(s')))

Using lemma read_mem_cnt_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `read_mem_cnt_TCC3.2`.
Q.E.D.

C.191.11 Random_Device.read_seed_TCC1

Terse proof for `read_seed_TCC1`.
`read_seed_TCC1`:

{1} ∀ (a: Address, bl: list[Byte]): cons?(bl) ∧ a = rnd_seed ⊃ (∀ (s: Random_device_memory): non_bool_integral?(uint))

Repeatedly Skolemizing and flattening,
Expanding the definition of non_bool_integral?,
Expanding the definition of unsigned_integer?,
which is trivially true.
This completes the proof of `read_seed_TCC1`.
Q.E.D.

C.191.12 Random_Device.read_seed_TCC2

Terse proof for `read_seed_TCC2`.
`read_seed_TCC2`:

{1} ∀ (a: Address, bl: list[Byte]): cons?(bl) ∧ a = rnd_seed ⊃ (∀ (s: Random_device_memory): max_value(uint) ≥ 0 ∧ max_value(uint) > 0)

Repeatedly Skolemizing and flattening,
Keeping (1) and hiding *,
Using lemma max_uint,
Using lemma max_uint_bits,
Using lemma pos_expt_gt,
Expanding the definition of max_value,
Expanding the definition of range_integral,
Case splitting on extend[real, int, bool, FALSE] (extend[int, nat, bool, FALSE](range_uint)) = extend[real, nat, bool, FALSE](range_uint),
we get 2 subgoals:

`read_seed_TCC2.1`:

```
{-1}  extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
        extend[real, nat, bool, FALSE](range_uint)
{-2}  max_value_bits_uint < 2 ^ max_value_bits_uint
{-3}  max_value_bits_uint ≥ 16
{-4}  max(range_uint) ≥ (2 ^ max_value_bits_uint) − 1
|-------
{1}   max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) ≥ 0 ∧
        max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) > 0
```

Replacing using formula -1,

Using lemma max_stable_under_extension[real, nat, ≤],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC2.1`.

`read_seed_TCC2.2`:

```
{-1}  max_value_bits_uint < 2 ^ max_value_bits_uint
{-2}  max_value_bits_uint ≥ 16
{-3}  max(range_uint) ≥ (2 ^ max_value_bits_uint) − 1
|-------
{1}   extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
        extend[real, nat, bool, FALSE](range_uint)
{2}   max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) ≥ 0 ∧
        max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) > 0
```

Keeping (1) and hiding *,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `read_seed_TCC2.2`.

Q.E.D.

C.191.13 Random_Device.read_seed_TCC3

Terse proof for `read_seed_TCC3`.

`read_seed_TCC3`:

```
|-------
{1}   ∀ (a: Address, bl: list[Byte]):
        cons?(bl) ∧ a = rnd_seed ⊃
          (∀ (s: Random_device_memory): (range_uint)(rem(max_value(uint))(seed(s))))
```

Repeatedly Skolemizing and flattening,

Using lemma read_seed_TCC2,

Using lemma binary_range_uint,

we get 2 subgoals:

`read_seed_TCC3.1:`

{-1}	range_uint(rem(max_value(uint))(seed(s'))) ≡ rem(max_value(uint))(seed(s')) ≤ max(range_uint)
{-2}	cons?(bl$'$) ∧ a' = rnd_seed ⊃ (∀ (s: Random_device_memory): max_value(uint) ≥ 0 ∧ max_value(uint) > 0)
{-3}	every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl$'$)
{-4}	cons?(bl$'$)
{-5}	a' = rnd_seed
{1}	(range_uint)(rem(max_value(uint))(seed(s')))

Adding type constraints for rem(max_value(uint))(seed(s!1)),

we get 2 subgoals:

`read_seed_TCC3.1.1:`

{-1}	rem(max_value(uint))(seed(s')) < max_value(uint)
{-2}	∃ q: seed(s') = rem(max_value(uint))(seed(s')) + max_value(uint) × q
{-3}	range_uint(rem(max_value(uint))(seed(s'))) ≡ rem(max_value(uint))(seed(s')) ≤ max(range_uint)
{-4}	cons?(bl$'$) ∧ a' = rnd_seed ⊃ (∀ (s: Random_device_memory): max_value(uint) ≥ 0 ∧ max_value(uint) > 0)
{-5}	every(λ (x: number): number_field_pred(x) ∧ real_pred(x) ∧ rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte) (bl$'$)
{-6}	cons?(bl$'$)
{-7}	a' = rnd_seed
{1}	(range_uint)(rem(max_value(uint))(seed(s')))

Expanding the definition of max_value,

Expanding the definition of range_integral,

Case splitting on extend[real, int, bool, FALSE] (extend[int, nat, bool, FALSE](range_uint)) = extend[real, nat, bool, FALSE](range_uint),

we get 2 subgoals:

`read_seed_TCC3.1.1.1:`

{-1} extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
extend[real, nat, bool, FALSE](range_uint)
{-2} rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
(seed(s'))
< max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)))
{-3} ∃ q:
seed(s') =
rem(max(extend [real, int, bool, FALSE](extend [int, nat, bool, FALSE](range_uint))))(seed(s')) + max
{-4} range_uint(rem(max(extend[real, int, bool, FALSE]
(extend[int, nat, bool, FALSE](range_uint))))
(seed(s')))
≡
rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
(seed(s'))
≤ max(range_uint)
{-5} cons?(bl′) ∧ a' = rnd_seed ⊃
(∀ (s: Random_device_memory):
max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) ≥ 0
∧
max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) >
0)
{-6} every(λ (x: number):
number_field_pred(x) ∧
real_pred(x) ∧
rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
(bl′)
{-7} cons?(bl′)
{-8} a' = rnd_seed

{1} (range_uint)(rem(max(extend[real, int, bool, FALSE]
(extend[int, nat, bool, FALSE](range_uint))))
(seed(s')))

Using lemma max_stable_under_extension[real, nat, ≤],

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC3.1.1.1`.

`read_seed_TCC3.1.1.2:`

```
{-1}  rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
        (seed(s'))
       < max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)))
{-2}  ∃ q:
        seed(s') =
         rem(max(extend [real, int, bool, FALSE](extend [int, nat, bool, FALSE](range_uint))))(seed
{-3}  range_uint(rem(max(extend[real, int, bool, FALSE]
                                  (extend[int, nat, bool, FALSE](range_uint))))
                       (seed(s')))
       ≡
       rem(max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))))
         (seed(s'))
        ≤ max(range_uint)
{-4}  cons?(bl') ∧ a' = rnd_seed ⊃
       (∀ (s: Random_device_memory):
          max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) ≥ 0
           ∧
           max(extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint))) >
            0)
{-5}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                 rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
         (bl')
{-6}  cons?(bl')
{-7}  a' = rnd_seed
------------------------------------------------------------------------
{1}   extend[real, int, bool, FALSE](extend[int, nat, bool, FALSE](range_uint)) =
       extend[real, nat, bool, FALSE](range_uint)
{2}   (range_uint)(rem(max(extend[real, int, bool, FALSE]
                                  (extend[int, nat, bool, FALSE](range_uint))))
                       (seed(s')))
```

Keeping (1) and hiding *,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `read_seed_TCC3.1.1.2`.
`read_seed_TCC3.1.2:`

```
{-1}  range_uint(rem(max_value(uint))(seed(s'))) ≡
       rem(max_value(uint))(seed(s')) ≤ max(range_uint)
{-2}  cons?(bl') ∧ a' = rnd_seed ⊃
       (∀ (s: Random_device_memory): max_value(uint) ≥ 0 ∧ max_value(uint) > 0)
{-3}  every(λ (x: number):
              number_field_pred(x) ∧
               real_pred(x) ∧
                 rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
         (bl')
{-4}  cons?(bl')
{-5}  a' = rnd_seed
------------------------------------------------------------------------
{1}   max_value(uint) ≥ 0
{2}   (range_uint)(rem(max_value(uint))(seed(s')))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC3.1.2`.

`read_seed_TCC3.2`:

```
{-1}  cons?(bl′) ∧ a′ = rnd_seed ⊃
       (∀ (s: Random_device_memory): max_value(uint) ≥ 0 ∧ max_value(uint) > 0)
{-2}  every(λ (x: number):
                number_field_pred(x) ∧
                 real_pred(x) ∧
                  rational_pred(x) ∧ integer_pred(x) ∧ x ≥ 0 ∧ x < max_byte)
            (bl′)
{-3}  cons?(bl′)
{-4}  a′ = rnd_seed
-------------------------------------------------------------------------------
{1}   max_value(uint) ≥ 0
{2}   (range_uint)(rem(max_value(uint))(seed(s′)))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `read_seed_TCC3.2`.

Q.E.D.

C.192 Proofs for Read_After_Write_Fails (datatype_model.pvs)

C.192.1 Read_After_Write_Fails.plain_mem_write_list_states_TCC1

Terse proof for `plain_mem_write_list_states_TCC1`.

`plain_mem_write_list_states_TCC1`:

```
-------------------------------------------------------------------------------
{1}   ∀ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
        size = length(bl) ∧
         plain_memory?(pm) ∧
          pm‘states(s) ∧
           (address_block(a, size) ⊆ pm‘rw_addr) ∧
            OK?(memory_write_list(pm‘mem)(a, bl)(s))
        ⊃
        OK?[State, Unit](memory_write_list[State](pm‘mem)(a, bl)(s)) ∨
         Exception?[State, Unit](memory_write_list[State](pm‘mem)(a, bl)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_write_list_states_TCC1`.

Q.E.D.

C.192.2 Read_After_Write_Fails.plain_mem_write_list_states

Terse proof for `plain_mem_write_list_states`.

`plain_mem_write_list_states:`

```
{1}  ∀ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
       size = length(bl) ∧
        plain_memory?(pm) ∧
         pm‘states(s) ∧
          (address_block(a, size) ⊆ pm‘rw_addr) ∧
           OK?(memory_write_list(pm‘mem)(a, bl)(s))
        ⊃ pm‘states(state(memory_write_list(pm‘mem)(a, bl)(s)))
```

Repeatedly Skolemizing and flattening,
Using lemma transformer_invariant_write_list[State],
Replacing using formula -4,
Using lemma plain_memory_side_effect_content_unchanged_write_block,
Replacing using formula -4,
Replacing using formula -3,
Replacing using formula -6,
Replacing using formula -1,
Rewriting using add_as_union, matching in *,
Rewriting using transformer_invariant_union_transformers, matching in *,
we get 3 subgoals:
`plain_mem_write_list_states.1:`

```
{-1}  side_effect_content_unchanged(address_block(a′, length(bl′)), pm′‘states,
                                                    memory_write_side_effect(pm′‘mem))
{-2}  result_pred(pm′‘states)(expr_2_super(memory_write_list(pm′‘mem)(a′, bl′))(s′))
{-3}  size′ = length(bl′)
{-4}  plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-7}  OK?(memory_write_list(pm′‘mem)(a′, bl′)(s′))
------------------------------------------------------------------------
{1}  pm′‘states(state(memory_write_list(pm′‘mem)(a′, bl′)(s′)))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_mem_write_list_states.1`.
`plain_mem_write_list_states.2:`

```
{-1}  side_effect_content_unchanged(address_block(a′, length(bl′)), pm′‘states,
                                                    memory_write_side_effect(pm′‘mem))
{-2}  transformer_invariant?(pm′‘states,
                                  (memory_write_transformers(pm′‘mem, address_block(a′, length(bl′))
       ⊃ result_pred(pm′‘states)(expr_2_super(memory_write_list(pm′‘mem)(a′, bl′))(s′))
{-3}  size′ = length(bl′)
{-4}  plain_memory?(pm′)
{-5}  pm′‘states(s′)
{-6}  (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-7}  OK?(memory_write_list(pm′‘mem)(a′, bl′)(s′))
------------------------------------------------------------------------
{1}  transformer_invariant?(pm′‘states,
                                  memory_write_transformers(pm′‘mem,
                                                                ad-
     dress_block(a′, length(bl′))))
{2}  pm′‘states(state(memory_write_list(pm′‘mem)(a′, bl′)(s′)))
```

Using lemma plain_memory_invariant_write_block,

Replacing using formula -5,

Replacing using formula -7,

which is trivially true.

This completes the proof of `plain_mem_write_list_states.2`.

`plain_mem_write_list_states.3`:

{-1} side_effect_content_unchanged(address_block(a', length(bl$'$)), pm$'$`states, memory_write_side_effect(pm$'$`mem))

{-2} transformer_invariant?(pm$'$`states, (memory_write_transformers(pm$'$`mem, address_block(a', length(bl$'$))) ∪ singleto
⊃ result_pred(pm$'$`states)(expr_2_super(memory_write_list(pm$'$`mem)(a', bl$'$))(s'))

{-3} size$'$ = length(bl$'$)

{-4} plain_memory?(pm$'$)

{-5} pm$'$`states(s')

{-6} (address_block(a', length(bl$'$)) ⊆ pm$'$`rw_addr)

{-7} OK?(memory_write_list(pm$'$`mem)(a', bl$'$)(s'))

{1} transformer_invariant?(pm$'$`states, singleton(expr_2_super(memory_write_side_effect(pm$'$`mem)(a', bl$'$, FALSE))))

{2} pm$'$`states(state(memory_write_list(pm$'$`mem)(a', bl$'$)(s')))

Using lemma plain_memory_transformer_invariant_write_side_effects,

Replacing using formula -5,

Replacing using formula -7,

Applying transformer_invariant_mono_transformers where transformers_1 gets singleton(expr_2_super(memory_write_side_ef (a!1, bl!1, FALSE))), transformers_2 gets memory_write_side_effect_super_transformers(pm!1`mem, address_block (a!1, length(bl!1))),

Instantiating quantified variables,

Replacing using formula -2,

Hiding formulas: -2, -3, -4, 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: 2,

Expanding the definition of subset?,

Expanding the definition of memory_write_side_effect_super_transformers,

Expanding the definition of expr_2_super,

Expanding the definition of singleton,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_write_list_states.3`.

Q.E.D.

C.192.3 Read_After_Write_Fails.plain_mem_write_list_read_list_ok

Terse proof for `plain_mem_write_list_read_list_ok`.

`plain_mem_write_list_read_list_ok:`

```
{1}  ∀ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
       size = length(bl) ∧
        plain_memory?(pm) ∧
         pm‘states(s) ∧
          (address_block(a, size) ⊆ pm‘rw_addr) ∧
           OK?(memory_write_list(pm‘mem)(a, bl)(s))
       ⊃
       OK?(memory_read_list(pm‘mem)(a, size)
                            (state(memory_write_list(pm‘mem)(a, bl)(s))))
```

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_ok[State],
Using lemma plain_memory_transformer_invariant_read_list_block,
Replacing using formula -4,
Applying subset_bigger_union_right where a gets address_block(a!1, size!1), b gets pm!1‘ro_addr, c gets pm!1‘rw_addr,
Replacing using formula -7,
Replacing using formula -1,
Replacing using formula -2,
Using lemma plain_memory_transformers_ok_read_list_block,
Replacing using formula -6,
Replacing using formula -2,
Replacing using formula -1,
Using lemma plain_mem_write_list_states,
Replacing using formula -6,
Replacing using formula -7,
Replacing using formula -8,
Replacing using formula -9,
Replacing using formula -10,
Replacing using formula -1,
which is trivially true.
This completes the proof of `plain_mem_write_list_read_list_ok`.
Q.E.D.

C.192.4 Read_After_Write_Fails.Fatal_Result

Terse proof for `Fatal_Result`.
`Fatal_Result:`

```
{1}  ∀ (a: Address, pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧ pm‘states(s) ∧ in_blessed_memory?(pod_1, a, pm‘rw_addr) ⊃
        Fatal?((write_data(pm, pod_1)(a, unit) ## read_data(pm, pod_2)(a))(s))
```

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: write_data in_blessed_memory? pod_1 pod_2 length read_data uidt_1 uidt_2 ##
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma memory_write_list_ok[State],
Using lemma plain_memory_transformer_invariant_write_list_block,
Using lemma plain_memory_transformers_ok_write_list_block,

Using lemma plain_memory_side_effect_content_unchanged_write_block,
Replacing using formula -5,
Replacing using formula -6,
Expanding the definition of length,
Expanding the definition of length,
Replacing using formula -1,
Replacing using formula -2,
Replacing using formula -3,
Using lemma plain_mem_write_list_read_list_ok,
Using lemma plain_mem_write_list_read_list,
Expanding the definition of ok_lift,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -6,
Keeping (-10) and hiding *,
Using lemma list_cons_eta,
Using lemma list_cons_eta,
Replacing using formula -2,
Replacing using formula -1,
Keeping (-3) and hiding *,
Applying decompose-equality,
This completes the proof of `Fatal_Result`.
Q.E.D.

C.193 Proofs for Read_After_Write_Fails_2 (datatype_model.pvs)

C.193.1 Read_After_Write_Fails_2.plain_mem_write_list_states_TCC1

Terse proof for `plain_mem_write_list_states_TCC1`.
`plain_mem_write_list_states_TCC1`:

```
{1}  ∀ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
       size = length(bl) ∧
        plain_memory?(pm) ∧
         pm`states(s) ∧
          (address_block(a, size) ⊆ pm`rw_addr) ∧
           OK?(memory_write_list(pm`mem)(a, bl)(s))
       ⊃
       OK?[State, Unit](memory_write_list[State](pm`mem)(a, bl)(s)) ∨
        Exception?[State, Unit](memory_write_list[State](pm`mem)(a, bl)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_mem_write_list_states_TCC1`.
Q.E.D.

C.193.2 Read_After_Write_Fails_2.plain_mem_write_list_states

Terse proof for `plain_mem_write_list_states`.

`plain_mem_write_list_states:`

{1} ∀ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
size = length(bl) ∧
plain_memory?(pm) ∧
pm‘states(s) ∧
(address_block(a, size) ⊆ pm‘rw_addr) ∧
OK?(memory_write_list(pm‘mem)(a, bl)(s))
⊃ pm‘states(state(memory_write_list(pm‘mem)(a, bl)(s)))

Repeatedly Skolemizing and flattening,
Using lemma transformer_invariant_write_list[State],
Replacing using formula -4,
Using lemma plain_memory_side_effect_content_unchanged_write_block,
Replacing using formula -4,
Replacing using formula -3,
Replacing using formula -6,
Replacing using formula -1,
Rewriting using add_as_union, matching in *,
Rewriting using transformer_invariant_union_transformers, matching in *,
we get 3 subgoals:
`plain_mem_write_list_states.1:`

{-1} side_effect_content_unchanged(address_block(a′, length(bl′)), pm′‘states,
memory_write_side_effect(pm′‘mem))
{-2} result_pred(pm′‘states)(expr_2_super(memory_write_list(pm′‘mem)(a′, bl′))(s′))
{-3} size′ = length(bl′)
{-4} plain_memory?(pm′)
{-5} pm′‘states(s′)
{-6} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-7} OK?(memory_write_list(pm′‘mem)(a′, bl′)(s′))

{1} pm′‘states(state(memory_write_list(pm′‘mem)(a′, bl′)(s′)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `plain_mem_write_list_states.1`.
`plain_mem_write_list_states.2:`

{-1} side_effect_content_unchanged(address_block(a′, length(bl′)), pm′‘states,
memory_write_side_effect(pm′‘mem))
{-2} transformer_invariant?(pm′‘states,
(memory_write_transformers(pm′‘mem, address_block(a′, length(bl′)))
⊃ result_pred(pm′‘states)(expr_2_super(memory_write_list(pm′‘mem)(a′, bl′))(s′))
{-3} size′ = length(bl′)
{-4} plain_memory?(pm′)
{-5} pm′‘states(s′)
{-6} (address_block(a′, length(bl′)) ⊆ pm′‘rw_addr)
{-7} OK?(memory_write_list(pm′‘mem)(a′, bl′)(s′))

{1} transformer_invariant?(pm′‘states,
memory_write_transformers(pm′‘mem,
address_block(a′, length(bl′))))
{2} pm′‘states(state(memory_write_list(pm′‘mem)(a′, bl′)(s′)))

Using lemma plain_memory_invariant_write_block,

Replacing using formula -5,

Replacing using formula -7,

which is trivially true.

This completes the proof of `plain_mem_write_list_states.2`.

`plain_mem_write_list_states.3`:

```
{-1}  side_effect_content_unchanged(address_block(a', length(bl')), pm'`states,
                                                    memory_write_side_effect(pm'`mem))
{-2}  transformer_invariant?(pm'`states,
                                  (memory_write_transformers(pm'`mem, address_block(a', length(bl'))) ∪ singleto
       ⊃ result_pred(pm'`states)(expr_2_super(memory_write_list(pm'`mem)(a', bl'))(s'))
{-3}  size' = length(bl')
{-4}  plain_memory?(pm')
{-5}  pm'`states(s')
{-6}  (address_block(a', length(bl')) ⊆ pm'`rw_addr)
{-7}  OK?(memory_write_list(pm'`mem)(a', bl')(s'))
-------------------------------------------------------------------------------------------
{1}   transformer_invariant?(pm'`states,
                                  singleton(expr_2_super(memory_write_side_effect(pm'`mem)
                                                                                  (a', bl', FALSE))))
{2}   pm'`states(state(memory_write_list(pm'`mem)(a', bl')(s')))
```

Using lemma plain_memory_transformer_invariant_write_side_effects,

Replacing using formula -5,

Replacing using formula -7,

Applying transformer_invariant_mono_transformers where transformers_1 gets singleton(expr_2_super(memory_write_side_ef
(a!1, bl!1, FALSE))), transformers_2 gets memory_write_side_effect_super_transformers(pm!1`mem, ad-
dress_block (a!1, length(bl!1))),

Instantiating quantified variables,

Replacing using formula -2,

Hiding formulas: -2, -3, -4, 2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Hiding formulas: 2,

Expanding the definition of subset?,

Expanding the definition of memory_write_side_effect_super_transformers,

Expanding the definition of expr_2_super,

Expanding the definition of singleton,

Expanding the definition of member,

Repeatedly Skolemizing and flattening,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `plain_mem_write_list_states.3`.

Q.E.D.

C.193.3 Read_After_Write_Fails_2.plain_mem_write_list_read_list_ok

Terse proof for `plain_mem_write_list_read_list_ok`.

plain_mem_write_list_read_list_ok:

{1} ∀ (a: Address, pm: Plain_Memory[State], s: State, size: nat, bl: list[Byte]):
size = length(bl) ∧
plain_memory?(pm) ∧
pm‘states(s) ∧
(address_block(a, size) ⊆ pm‘rw_addr) ∧
OK?(memory_write_list(pm‘mem)(a, bl)(s))
⊃
OK?(memory_read_list(pm‘mem)(a, size)
(state(memory_write_list(pm‘mem)(a, bl)(s))))

Repeatedly Skolemizing and flattening,
Using lemma memory_read_list_ok[State],
Using lemma plain_memory_transformer_invariant_read_list_block,
Replacing using formula -4,
Applying subset_bigger_union_right where a gets address_block(a!1, size!1), b gets pm!1‘ro_addr, c gets pm!1‘rw_addr,
Replacing using formula -7,
Replacing using formula -1,
Replacing using formula -2,
Using lemma plain_memory_transformers_ok_read_list_block,
Replacing using formula -6,
Replacing using formula -2,
Replacing using formula -1,
Using lemma plain_mem_write_list_states,
Replacing using formula -6,
Replacing using formula -7,
Replacing using formula -8,
Replacing using formula -9,
Replacing using formula -10,
Replacing using formula -1,
which is trivially true.
This completes the proof of plain_mem_write_list_read_list_ok.
Q.E.D.

C.193.4 Read_After_Write_Fails_2.Fatal_Result

Terse proof for Fatal_Result.
Fatal_Result:

{1} ∀ (a: Address, pm: Plain_Memory[State], s: State):
plain_memory?(pm) ∧ pm‘states(s) ∧ in_blessed_memory?(pod_bool, a, pm‘rw_addr) ⊃
Fatal?((write_data(pm, pod_bool)(a, TRUE) ## read_data(pm, pod_Byte)(a))(s))

Repeatedly Skolemizing and flattening,
Expanding the definition of write_data,
Expanding the definition of write_data,
Expanding the definition of in_blessed_memory?,
Expanding the definition of pod_bool,
Expanding the definition of ##,
Expanding the definition of ##,

Using lemma memory_write_list_ok[State],
Using lemma plain_memory_transformer_invariant_write_list_block,
Expanding the definition of uidt_bool,
Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
Replacing using formula -3,
Replacing using formula -5,
Replacing using formula -1,
Using lemma plain_memory_transformers_ok_write_list_block,
Replacing using formula -4,
Replacing using formula -6,
Replacing using formula -1,
Using lemma plain_memory_side_effect_content_unchanged_write_block,
Replacing using formula -5,
Replacing using formula -7,
Replacing using formula -1,
Replacing using formula -6,
Simplifying, rewriting, and recording with decision procedures,
Expanding the definition of read_data,
Expanding the definition of read_data,
Expanding the definition of pod_Byte,
Expanding the definition of uidt_Byte,
Expanding the definition of ##,
Using lemma plain_mem_write_list_read_list_ok,
Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
Simplifying, rewriting, and recording with decision procedures,
Using lemma plain_mem_write_list_read_list,
Expanding the definition of length,
Expanding the definition of length,
Expanding the definition of length,
Replacing using formula -1,
Expanding the definition of ok_lift,
which is trivially true.
This completes the proof of `Fatal_Result`.
Q.E.D.

C.194 Proofs for Register_Id (constants.pvs)

This theory contains no provable formal statements.

C.195 Proofs for Register_Id_adt (Register_Id_adt.pvs)

This theory contains no provable formal statements.

C.196 Proofs for Register_Id_adt_reduce (Register_Id_adt.pvs)

This theory contains no provable formal statements.

C.197 Proofs for Rewrite_Test (plain_memory_rewrites.pvs)

C.197.1 Rewrite_Test.rewrite_test

Terse proof for `rewrite_test`.

`rewrite_test:`

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]
        dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
     plain_memory?(pm) ∧
      in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧
       in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
        blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
         blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm`states(s)
      ⊃
      OK?((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
             read_data(pm, dt1)(addr1)
             ## write_data(pm, dt2)(addr2, data2)
             ## read_data(pm, dt2)(addr2)
             ## read_data(pm, dt1)(addr1))
              (s))
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_s pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test`.

Q.E.D.

C.197.2 Rewrite_Test.rewrite_test_data_TCC1

Terse proof for `rewrite_test_data_TCC1`.

`rewrite_test_data_TCC1:`

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       pm‘states(s) ∧
        blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧
         blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
          in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
           in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧ plain_memory?(pm)
        ⊃
        OK?[State, Data1]
            ((##[State, Data2, Data1]
                    (##[State, Unit, Data2]
                           (##[State, Data1, Unit]
                                  (##[State, Unit, Data1]
                                         (##[State, Unit, Unit]
                                                (write_data[State, Data1](pm, dt1)(addr1, data1),
                                                  write_data[State, Data2](pm, dt2)(addr2, data2)),
                                           read_data[State, Data1](pm, dt1)(addr1)),
                                    write_data[State, Data2](pm, dt2)(addr2, data2)),
                             read_data[State, Data2](pm, dt2)(addr2)),
                      read_data[State, Data1](pm, dt1)(addr1)))
                (s))
```

Repeatedly Skolemizing and flattening,
Rewriting using rewrite_test, matching in *,
This completes the proof of `rewrite_test_data_TCC1`.
Q.E.D.

C.197.3 Rewrite_Test.rewrite_test_data

Terse proof for `rewrite_test_data`.

`rewrite_test_data:`

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
         in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm‘states(s)
        ⊃
        data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
                  read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## read_data(pm, dt1)(addr1))
                   (s))
         = data1
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr

plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read
pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing
plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res
pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st
pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr
plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt
pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr
plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data
pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin
pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s
pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data`.

Q.E.D.

C.197.4 Rewrite_Test.rewrite_test_data_long_TCC1

Terse proof for `rewrite_test_data_long_TCC1`.

rewrite_test_data_long_TCC1:

{1} ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
pm`states(s) ∧
blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧
blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧ plain_memory?(pm)
⊃
OK?[State, Data1]
((##[State, Data2, Data1]
(##[State, Unit, Data2]
(##[State, Data1, Unit]
(##[State, Unit, Data1]
(##[State, Data2, Unit]
(##[State, Unit, Data2]
(##[State, Data1, Unit]
(##[State, Unit, Data1]
(##[State, Data2, Unit]
(##[State, Unit, Data2]
(##[State, Data1, Unit]
(##[State, Unit, Data1]
(##[State, Data2, Unit]
(##[State, Unit, Data2
(##[State, Data1, U
(##
[State, Unit, Dat
(##
[State,
Data2,
Unit]
(##
[State,
Unit,
Data2]
(##
[State,
Data1,
Unit]
(##
[State,
Unit,
Data1]
(##
[State,
Data
Unit
(##
[Sta
U
D
(##
[S

2881

(#

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_s pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_long_TCC1`.

Q.E.D.

C.197.5 Rewrite_Test.rewrite_test_data_long

Terse proof for `rewrite_test_data_long`.

`rewrite_test_data_long:`

```
  |-------
{1}   ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
        plain_memory?(pm) ∧
         in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧
          in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
           blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
            blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm`states(s)
         ⊃
         data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
                  read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## read_data(pm, dt1)(addr1))
                   (s))
          = data1
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr

plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_long`.

Q.E.D.

C.197.6 Rewrite_Test.rewrite_test_data_eval_if_ok_ok

Terse proof for `rewrite_test_data_eval_if_ok_ok`.

`rewrite_test_data_eval_if_ok_ok`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]
         dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
         in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm‘states(s)
        ⊃
        OK?((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
               read_data(pm, dt1)(addr1)
               ## write_data(pm, dt2)(addr2, data2)
               ## read_data(pm, dt2)(addr2)
               ## (λ (d: Data2): read_data(pm, dt1)(addr1)))
                (s))
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_eval_if_ok_ok`.

Q.E.D.

C.197.7 Rewrite_Test.rewrite_test_data_eval_if_ok_TCC1

Terse proof for `rewrite_test_data_eval_if_ok_TCC1`.

`rewrite_test_data_eval_if_ok_TCC1`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       pm`states(s) ∧
        blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧
         blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
          in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
           in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧ plain_memory?(pm)
        ⊃
        OK?[State, Data1]
            ((##[State, Data2, Data1]
                    (##[State, Unit, Data2]
                            (##[State, Data1, Unit]
                                    (##[State, Unit, Data1]
                                            (##[State, Unit, Unit]
                                                    (write_data[State, Data1](pm, dt1)(addr1, data1),
                                                      write_data[State, Data2](pm, dt2)(addr2, data2)),
                                                read_data[State, Data1](pm, dt1)(addr1)),
                                            write_data[State, Data2](pm, dt2)(addr2, data2)),
                                    read_data[State, Data2](pm, dt2)(addr2)),
                            λ (d: Data2): read_data[State, Data1](pm, dt1)(addr1)))
                    (s))
```

Repeatedly Skolemizing and flattening,
Rewriting using rewrite_test_data_eval_if_ok_ok, matching in *,
This completes the proof of `rewrite_test_data_eval_if_ok_TCC1`.
Q.E.D.

C.197.8 Rewrite_Test.rewrite_test_data_eval_if_ok

Terse proof for `rewrite_test_data_eval_if_ok`.

`rewrite_test_data_eval_if_ok`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
          dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧
         in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm`states(s)
        ⊃
        data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
                  read_data(pm, dt1)(addr1)
                  ## write_data(pm, dt2)(addr2, data2)
                  ## read_data(pm, dt2)(addr2)
                  ## (λ (d: Data2): read_data(pm, dt1)(addr1)))
                   (s))
         = data1
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_eval_if_ok`.

Q.E.D.

C.197.9 Rewrite_Test.rewrite_test_data_eval_if_ok_ok_result_ok

Terse proof for `rewrite_test_data_eval_if_ok_ok_result_ok`.

`rewrite_test_data_eval_if_ok_ok_result_ok`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]
         dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, pm‘rw_addr) ∧
         in_blessed_memory?(dt2, addr2, pm‘rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm‘states(s)
        ⊃
        OK?[State, Data2]
            ((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
                 read_data(pm, dt1)(addr1)
                 ## write_data(pm, dt2)(addr2, data2)
                 ## read_data(pm, dt2)(addr2)
                 ## (λ (d: Data2): ok_result(d)))
              (s))
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_st pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_eval_if_ok_ok_result_ok`.

Q.E.D.

C.197.10 Rewrite_Test.rewrite_test_data_eval_if_ok_ok_result_TCC1

Terse proof for `rewrite_test_data_eval_if_ok_ok_result_TCC1`.

`rewrite_test_data_eval_if_ok_ok_result_TCC1`:

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]),
         dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       pm`states(s) ∧
        blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧
         blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
          in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
           in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧ plain_memory?(pm)
        ⊃
        OK?[State, Data2]
            ((##[State, Data2, Data2]
                    (##[State, Unit, Data2]
                            (##[State, Data1, Unit]
                                    (##[State, Unit, Data1]
                                            (##[State, Unit, Unit]
                                                    (write_data[State, Data1](pm, dt1)(addr1, data1),
                                                      write_data[State, Data2](pm, dt2)(addr2, data2)),
                                                  read_data[State, Data1](pm, dt1)(addr1)),
                                                write_data[State, Data2](pm, dt2)(addr2, data2)),
                                            read_data[State, Data2](pm, dt2)(addr2)),
                                        λ (d: Data2): ok_result[State, Data2](d)))
                                (s))
```

Repeatedly Skolemizing and flattening,

Rewriting using rewrite_test_data_eval_if_ok_ok_result_ok, matching in *,

This completes the proof of `rewrite_test_data_eval_if_ok_ok_result_TCC1`.

Q.E.D.

C.197.11 Rewrite_Test.rewrite_test_data_eval_if_ok_ok_result

Terse proof for `rewrite_test_data_eval_if_ok_ok_result`.

`rewrite_test_data_eval_if_ok_ok_result:`

```
{1}  ∀ (addr1, addr2: Address, data1: Data1, data2: Data2, dt1: (interpreted_data_type?[Data1]
         dt2: (interpreted_data_type?[Data2]), pm: Plain_Memory[State], s: State):
       plain_memory?(pm) ∧
        in_blessed_memory?(dt1, addr1, pm`rw_addr) ∧
         in_blessed_memory?(dt2, addr2, pm`rw_addr) ∧
          blocks_disjoint?(addr1, size(uidt(dt1)), addr2, size(uidt(dt2))) ∧
           blocks_disjoint?(addr2, size(uidt(dt2)), addr1, size(uidt(dt1))) ∧ pm`states(s)
        ⊃
        data((write_data(pm, dt1)(addr1, data1) ## write_data(pm, dt2)(addr2, data2) ##
                read_data(pm, dt1)(addr1)
                ## write_data(pm, dt2)(addr2, data2)
                ## read_data(pm, dt2)(addr2)
                ## (λ (d: Data2): ok_result(d)))
                 (s))
         = data2
```

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_sing plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_s pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_expr plain_memory_read_read_q_data pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_sin pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: comp_eval_if_ok_fexpr ok_result_q_ok ok_result_ok ok_result_q_data different_registers_blocks_disjoint

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `rewrite_test_data_eval_if_ok_ok_result`.

Q.E.D.

C.198 Proofs for Search_Example (search-example.pvs)

C.198.1 Search_Example.value_TCC1

Terse proof for `value_TCC1`.

`value_TCC1:`

```
{1}  ∃ (x: (range(uint))): TRUE
```

Instantiating the top quantifier in 1 with the terms: (0),

we get 2 subgoals:

`value_TCC1.1:`

```
{1}  TRUE
```

which is trivially true.
This completes the proof of `value_TCC1.1`.
`value_TCC1.2`:

{1} range(uint)(0)

Trying repeated skolemization, instantiation, and if-lifting,
Rewriting using binary_range_uint, matching in *,
Using lemma max_uint,
Using lemma max_uint_bits,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `value_TCC1.2`.
Q.E.D.

C.198.2 Search_Example.Search_Pre_TCC1

Terse proof for `Search_Pre_TCC1`.
`Search_Pre_TCC1`:

{1} Cpp_Type?(uint) ∧ cv(non_bool_integral_enum?)(uint)

Applying propositional simplification,
we get 2 subgoals:
`Search_Pre_TCC1.1`:

{1} Cpp_Type?(uint)

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?
Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `Search_Pre_TCC1.1`.
`Search_Pre_TCC1.2`:

{1} cv(non_bool_integral_enum?)(uint)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Search_Pre_TCC1.2`.
Q.E.D.

C.198.3 Search_Example.Search_Pre_TCC2

Terse proof for `Search_Pre_TCC2`.

`Search_Pre_TCC2:`

{1} ∀ (s: State):
pm‘states(s) ∧ plain_memory?(pm) ⊃
Cpp_Type?(pointer(uint)) ∧ cv(pointer?)(pointer(uint))

Repeatedly Skolemizing and flattening,

Keeping (1) and hiding *,

Applying propositional simplification,

we get 2 subgoals:

`Search_Pre_TCC2.1:`

{1} Cpp_Type?(pointer(uint))

Expanding the definition of Cpp_Type?,

Expanding the definition of subterm,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Splitting conjunctions,

we get 2 subgoals:

`Search_Pre_TCC2.1.1:`

{-1} $t' = \text{pointer(uint)}$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge$
$\text{no\_pointers\_to\_references?}(t') \wedge$
$\text{no\_cv\_references?}(t') \wedge$
$\text{no\_reference\_to\_reference?}(t') \wedge$
$\text{no\_reference\_to\_bitfields?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge$
$\text{no\_cv\_void\_parameter?}(t') \wedge$
$\text{no\_array\_of\_references?}(t') \wedge$
$\text{no\_array\_of\_cv\_void?}(t') \wedge$
$\text{no\_array\_of\_function?}(t') \wedge$
$\text{no\_array\_of\_abstract\_class?}(t') \wedge$
$\text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge$
$\text{no\_array\_return\_type?}(t') \wedge$
$\text{no\_function\_return\_type?}(t') \wedge$
$\text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge$
$\text{cv\_array?}(t') \wedge$
$\text{enum\_underlying\_integral?}(t') \wedge$
$\text{enum\_constants?}(t') \wedge$
$\text{const\_volatile?}(t') \wedge$
$\text{const\_stutter?}(t') \wedge$
$\text{volatile\_stutter?}(t') \wedge$
$\text{no\_cv\_class?}(t') \wedge$
$\text{no\_cv\_union?}(t') \wedge$
$\text{no\_cv\_function?}(t') \wedge$
$\text{no\_reference\_to\_void?}(t') \wedge$
$\text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `Search_Pre_TCC2.1.1`.

`Search_Pre_TCC2.1.2:`

{-1} $t' = \text{uint}$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `Search_Pre_TCC2.1.2`.
`Search_Pre_TCC2.2:`

{1} cv(pointer?)(pointer(uint))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `Search_Pre_TCC2.2`.
Q.E.D.

C.198.4 Search_Example.Search_Pre_TCC3

Terse proof for `Search_Pre_TCC3`.
`Search_Pre_TCC3:`

{1} $\forall\ (s\text{: State})\text{: } \text{pm`states}(s) \wedge \text{plain\_memory?}(\text{pm}) \supset \text{Cpp\_Type?}(\text{pointer}(\text{uint}))$

Repeatedly Skolemizing and flattening,
Using lemma Search_Pre_TCC2,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `Search_Pre_TCC3`.

Q.E.D.

C.198.5 Search_Example.Search_Pre_TCC4

Terse proof for `Search_Pre_TCC4`.

`Search_Pre_TCC4:`

{1} ∀ (s: State): pm‘states(s) ∧ plain_memory?(pm) ⊃ pointer?(cv_base(pointer(uint)))

Repeatedly Skolemizing and flattening,

Rewriting using cv_base_result, matching in *,

we get 2 subgoals:

`Search_Pre_TCC4.1:`

{-1}	pm‘states(s′)
{-2}	plain_memory?(pm)
{1}	cv(pointer?)(pointer(uint))
{2}	pointer?(cv_base(pointer(uint)))

Using lemma Search_Pre_TCC2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `Search_Pre_TCC4.1`.

`Search_Pre_TCC4.2:`

{-1}	pm‘states(s′)
{-2}	plain_memory?(pm)
{1}	(pointer? ⊆ interpreted?)
{2}	pointer?(cv_base(pointer(uint)))

Keeping (1) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `Search_Pre_TCC4.2`.

Q.E.D.

C.198.6 Search_Example.Search_Pre_TCC5

Terse proof for `Search_Pre_TCC5`.

Search_Pre_TCC5:

{1} ∀ (s: State):
blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm‘rw_addr) ∧
valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
(pm‘ro_addr ∪ pm‘rw_addr))
∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
(pm‘ro_addr ∪ pm‘rw_addr))
∧ pm‘states(s) ∧ plain_memory?(pm)
⊃
OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
(read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
(pm[State], dt_cv_pointer(pointer(uint)))(first)(s))

Repeatedly Skolemizing and flattening,
Rewriting using plain_memory_read_data_ok, matching in *,
This completes the proof of Search_Pre_TCC5.
Q.E.D.

C.198.7 Search_Example.Search_Pre_TCC6

Terse proof for Search_Pre_TCC6.

Search_Pre_TCC6:

{1} ∀ (s: State):
blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm‘rw_addr) ∧
valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
(pm‘ro_addr ∪ pm‘rw_addr))
∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
(pm‘ro_addr ∪ pm‘rw_addr))
∧ pm‘states(s) ∧ plain_memory?(pm)
⊃
(∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
(read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
(pm[State], dt_cv_pointer(pointer(uint)))(last)(s)))

Repeatedly Skolemizing and flattening,

Rewriting using plain_memory_read_data_ok, matching in *,

This completes the proof of `Search_Pre_TCC6`.

Q.E.D.

C.198.8 Search_Example.Search_Pre_TCC7

Terse proof for `Search_Pre_TCC7`.

`Search_Pre_TCC7`:

```
{1}  ∀ (s: State):
       blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
        blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
         blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
          blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
           in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm‘rw_addr) ∧
            valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
             valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
              in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
                                 (pm‘ro_addr ∪ pm‘rw_addr))
               ∧
               in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
                                  (pm‘ro_addr ∪ pm‘rw_addr))
                ∧ pm‘states(s) ∧ plain_memory?(pm)
       ⊃
       (∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
            first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
             (∀ (last_value: ((range_pointer(cv_base(pointer(uint)))))):
                  not_null?(pointer(uint))(last_value) ∧
                   not_null?(pointer(uint))(first_value) ∧
                    last_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s))
                   ⊃ (∀ (i: below[N]): Cpp_Type?(uint))))
```

Repeatedly Skolemizing and flattening,

Using lemma Search_Pre_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `Search_Pre_TCC7`.

Q.E.D.

C.198.9 Search_Example.Search_Pre_TCC8

Terse proof for `Search_Pre_TCC8`.

`Search_Pre_TCC8:`

```
{1}  ∀ (s: State):
       blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
        blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
         blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
          blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
           in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm‘rw_addr) ∧
            valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
             valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
              in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
                                 (pm‘ro_addr ∪ pm‘rw_addr))
               ∧
               in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
                                  (pm‘ro_addr ∪ pm‘rw_addr))
                ∧ pm‘states(s) ∧ plain_memory?(pm)
       ⊃
       (∀ (first_value: ((range_pointer(cv_base(pointer(uint))))))):
            first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
             (∀ (last_value: ((range_pointer(cv_base(pointer(uint))))))):
                  not_null?(pointer(uint))(last_value) ∧
                   not_null?(pointer(uint))(first_value) ∧
                    last_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s))
                   ⊃
                   (∀ (i: below[N]):
                        up?[Memory_Address]
                            (address_of(pointer(uint))
                                       (add(pointer(uint))
                                           (first_value, i × size_of(uint)))))))
```

Repeatedly Skolemizing and flattening,

Rewriting using address_of_spec, matching in *,

Keeping (-17 1) and hiding *,

Using lemma add_spec,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `Search_Pre_TCC8`.

Q.E.D.

C.198.10 Search_Example.Search_Pre_TCC9

Terse proof for `Search_Pre_TCC9`.

Search_Pre_TCC9:

{1} ∀ (s: State):
blocks_disjoint?(current, size_of(pointer(uint)), last, size_of(pointer(uint))) ∧
blocks_disjoint?(last, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
blocks_disjoint?(current, size_of(pointer(uint)), first, size_of(pointer(uint))) ∧
blocks_disjoint?(first, size_of(pointer(uint)), current, size_of(pointer(uint))) ∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), current, pm‘rw_addr) ∧
valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(last)(s) ∧
valid_in_mem(pm, dt_cv_pointer(pointer(uint)))(first)(s) ∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), last,
(pm‘ro_addr ∪ pm‘rw_addr))
∧
in_blessed_memory?(dt_cv_pointer(pointer(uint)), first,
(pm‘ro_addr ∪ pm‘rw_addr))
∧ pm‘states(s) ∧ plain_memory?(pm)
⊃
(∀ (first_value: ((range_pointer(cv_base(pointer(uint)))))):
first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
(∀ (last_value: ((range_pointer(cv_base(pointer(uint)))))):
(∀ (i: below[N]):
valid_in_mem(pm, dt(uint))
(down(address_of(pointer(uint))
(add(pointer(uint))
(first_value,
i × size_of(uint)))))
(s))
∧
(∀ (i: below[N]):
blocks_disjoint?(current, size_of(pointer(uint)),
down(address_of(pointer(uint))
(add(pointer(uint))
(first_value,
i × size_of(uint)))),
size_of(uint)))
∧
(∀ (i: below[N]):
blocks_disjoint?(down(address_of(pointer(uint))
(add(pointer(uint))
(first_value,
i × size_of(uint)))),
size_of(uint), current, size_of(pointer(uint))))
∧
(∀ (i: below[N]):
in_blessed_memory?[(range(uint))]
(dt(uint),
down(address_of(pointer(uint))
(add(pointer(uint))
(first_value, i × size_of(uint)))),
(pm‘ro_addr ∪ pm‘rw_addr)))
∧
not_null?(pointer(uint))(last_value) ∧
not_null?(pointer(uint))(first_value) ∧
last_value =
data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s))
⊃ Cpp_Type?(uint)))

Repeatedly Skolemizing and flattening,

Using lemma Search_Pre_TCC1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `Search_Pre_TCC9`.

Q.E.D.

C.198.11 Search_Example.search_TCC1

Terse proof for `search_TCC1`.

`search_TCC1:`

{1} Cpp_Type?(pointer(uint)) ∧ cv(pointer?)(pointer(uint))

Applying propositional simplification,

we get 2 subgoals:

`search_TCC1.1:`

{1} Cpp_Type?(pointer(uint))

Expanding the definition of Cpp_Type?,

Repeatedly Skolemizing and flattening,

Expanding the definition of subterm,

Expanding the definition of subterm,

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Splitting conjunctions,

we get 2 subgoals:

`search_TCC1.1.1:`

{-1} $t' = \text{pointer(uint)}$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge \text{no\_pointers\_to\_references?}(t') \wedge \text{no\_cv\_references?}(t') \wedge \text{no\_reference\_to\_reference?}(t') \wedge \text{no\_reference\_to\_bitfields?}(t') \wedge \text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge \text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge \text{no\_cv\_void\_parameter?}(t') \wedge \text{no\_array\_of\_references?}(t') \wedge \text{no\_array\_of\_cv\_void?}(t') \wedge \text{no\_array\_of\_function?}(t') \wedge \text{no\_array\_of\_abstract\_class?}(t') \wedge \text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge \text{no\_array\_return\_type?}(t') \wedge \text{no\_function\_return\_type?}(t') \wedge \text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge \text{cv\_array?}(t') \wedge \text{enum\_underlying\_integral?}(t') \wedge \text{enum\_constants?}(t') \wedge \text{const\_volatile?}(t') \wedge \text{const\_stutter?}(t') \wedge \text{volatile\_stutter?}(t') \wedge \text{no\_cv\_class?}(t') \wedge \text{no\_cv\_union?}(t') \wedge \text{no\_cv\_function?}(t') \wedge \text{no\_reference\_to\_void?}(t') \wedge \text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Keeping (-2) and hiding *,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `search_TCC1.1.1`.

`search_TCC1.1.2:`

{-1} $t' = \text{uint}$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge$
$\text{no\_pointers\_to\_references?}(t') \wedge$
$\text{no\_cv\_references?}(t') \wedge$
$\text{no\_reference\_to\_reference?}(t') \wedge$
$\text{no\_reference\_to\_bitfields?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge$
$\text{no\_cv\_void\_parameter?}(t') \wedge$
$\text{no\_array\_of\_references?}(t') \wedge$
$\text{no\_array\_of\_cv\_void?}(t') \wedge$
$\text{no\_array\_of\_function?}(t') \wedge$
$\text{no\_array\_of\_abstract\_class?}(t') \wedge$
$\text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge$
$\text{no\_array\_return\_type?}(t') \wedge$
$\text{no\_function\_return\_type?}(t') \wedge$
$\text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge$
$\text{cv\_array?}(t') \wedge$
$\text{enum\_underlying\_integral?}(t') \wedge$
$\text{enum\_constants?}(t') \wedge$
$\text{const\_volatile?}(t') \wedge$
$\text{const\_stutter?}(t') \wedge$
$\text{volatile\_stutter?}(t') \wedge$
$\text{no\_cv\_class?}(t') \wedge$
$\text{no\_cv\_union?}(t') \wedge$
$\text{no\_cv\_function?}(t') \wedge$
$\text{no\_reference\_to\_void?}(t') \wedge$
$\text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `search_TCC1.1.2`.
`search_TCC1.2:`

{1} cv(pointer?)(pointer(uint))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `search_TCC1.2`.
Q.E.D.

C.198.12 Search_Example.search_TCC2

Terse proof for `search_TCC2`.
`search_TCC2:`

{1} Cpp_Type?(pointer(uint))

Using lemma search_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `search_TCC2`.
Q.E.D.

C.198.13 Search_Example.search_TCC3

Terse proof for `search_TCC3`.

`search_TCC3:`

|

{1} pointer?(cv_base(pointer(uint)))

Rewriting using cv_base_result, matching in *,
we get 2 subgoals:

`search_TCC3.1:`

|

{1} cv(pointer?)(pointer(uint))
{2} pointer?(cv_base(pointer(uint)))

Using lemma search_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `search_TCC3.1`.

`search_TCC3.2:`

|

{1} (pointer? $\subseteq$ interpreted?)
{2} pointer?(cv_base(pointer(uint)))

Hiding formulas: 2,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `search_TCC3.2`.
Q.E.D.

C.198.14 Search_Example.search_TCC4

Terse proof for `search_TCC4`.

`search_TCC4:`

|

{1} Cpp_Type?(pointer(uint)) $\wedge$ v(pointer?)(pointer(uint))

Applying propositional simplification,
we get 2 subgoals:

`search_TCC4.1:`

|

{1} Cpp_Type?(pointer(uint))

Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Expanding the definition of subterm,
Splitting conjunctions,

we get 2 subgoals:

`search_TCC4.1.1:`

{-1} $t' = \text{pointer(uint)}$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge$
$\text{no\_pointers\_to\_references?}(t') \wedge$
$\text{no\_cv\_references?}(t') \wedge$
$\text{no\_reference\_to\_reference?}(t') \wedge$
$\text{no\_reference\_to\_bitfields?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge$
$\text{no\_cv\_void\_parameter?}(t') \wedge$
$\text{no\_array\_of\_references?}(t') \wedge$
$\text{no\_array\_of\_cv\_void?}(t') \wedge$
$\text{no\_array\_of\_function?}(t') \wedge$
$\text{no\_array\_of\_abstract\_class?}(t') \wedge$
$\text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge$
$\text{no\_array\_return\_type?}(t') \wedge$
$\text{no\_function\_return\_type?}(t') \wedge$
$\text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge$
$\text{cv\_array?}(t') \wedge$
$\text{enum\_underlying\_integral?}(t') \wedge$
$\text{enum\_constants?}(t') \wedge$
$\text{const\_volatile?}(t') \wedge$
$\text{const\_stutter?}(t') \wedge$
$\text{volatile\_stutter?}(t') \wedge$
$\text{no\_cv\_class?}(t') \wedge$
$\text{no\_cv\_union?}(t') \wedge$
$\text{no\_cv\_function?}(t') \wedge$
$\text{no\_reference\_to\_void?}(t') \wedge$
$\text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

Replacing using formula -1,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `search_TCC4.1.1`.

search_TCC4.1.2:

{-1} $t' = \text{uint}$

{1} $\text{no\_pointers\_to\_bitfield?}(t') \wedge$
$\text{no\_pointers\_to\_references?}(t') \wedge$
$\text{no\_cv\_references?}(t') \wedge$
$\text{no\_reference\_to\_reference?}(t') \wedge$
$\text{no\_reference\_to\_bitfields?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_reference?}(t') \wedge$
$\text{no\_pointer\_to\_member\_to\_cv\_void?}(t') \wedge$
$\text{no\_cv\_void\_parameter?}(t') \wedge$
$\text{no\_array\_of\_references?}(t') \wedge$
$\text{no\_array\_of\_cv\_void?}(t') \wedge$
$\text{no\_array\_of\_function?}(t') \wedge$
$\text{no\_array\_of\_abstract\_class?}(t') \wedge$
$\text{no\_pointer\_or\_ref\_to\_incomplete\_array\_parameter?}(t') \wedge$
$\text{no\_array\_return\_type?}(t') \wedge$
$\text{no\_function\_return\_type?}(t') \wedge$
$\text{bitfield\_underlying\_integral\_or\_enum\_type?}(t') \wedge$
$\text{cv\_array?}(t') \wedge$
$\text{enum\_underlying\_integral?}(t') \wedge$
$\text{enum\_constants?}(t') \wedge$
$\text{const\_volatile?}(t') \wedge$
$\text{const\_stutter?}(t') \wedge$
$\text{volatile\_stutter?}(t') \wedge$
$\text{no\_cv\_class?}(t') \wedge$
$\text{no\_cv\_union?}(t') \wedge$
$\text{no\_cv\_function?}(t') \wedge$
$\text{no\_reference\_to\_void?}(t') \wedge$
$\text{no\_cv\_void?}(t') \wedge \text{no\_array\_of\_bitfields?}(t')$

Replacing using formula -1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of search_TCC4.1.2.

search_TCC4.2:

{1} $v(\text{pointer?})(\text{pointer}(\text{uint}))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of search_TCC4.2.
Q.E.D.

C.198.15 Search_Example.deref_current_TCC1

Terse proof for deref_current_TCC1.

deref_current_TCC1:

{1} $\forall\ (s\colon \text{State})\colon$
$\text{read\_current}(s) \supset$
$\text{OK?}[\text{State}, ((\text{range\_pointer}(\text{cv\_base}(\text{pointer}(\text{uint})))))]$
$(\text{read\_data}[\text{State}, ((\text{range\_pointer}(\text{cv\_base}(\text{pointer}(\text{uint})))))]$
$(\text{pm}[\text{State}], \text{dt\_cv\_pointer}(\text{pointer}(\text{uint})))(\text{current})(s))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `deref_current_TCC1`.
Q.E.D.

C.198.16 Search_Example.deref_current_TCC2

Terse proof for `deref_current_TCC2`.
`deref_current_TCC2`:

```
{1}   ∀ (s: State):
        read_current(s) ⊃
         (∀ (current_value: ((range_pointer(cv_base(pointer(uint))))):
               not_null?(pointer(uint))(current_value) ∧
                current_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))
                ⊃ up?[Memory_Address](address_of(pointer(uint))(current_value)))
```

Repeatedly Skolemizing and flattening,
Rewriting using address_of_spec, matching in *,
This completes the proof of `deref_current_TCC2`.
Q.E.D.

C.198.17 Search_Example.read_array_TCC1

Terse proof for `read_array_TCC1`.
`read_array_TCC1`:

```
{1}   ∀ (s: State):
        read_first(s) ⊃
         OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
               (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
                     (pm[State], dt_cv_pointer(pointer(uint)))(first)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of read_first,
which is trivially true.
This completes the proof of `read_array_TCC1`.
Q.E.D.

C.198.18 Search_Example.read_array_TCC2

Terse proof for `read_array_TCC2`.
`read_array_TCC2`:

```
{1}   ∀ (s: State):
        read_first(s) ⊃
         (∀ (first_value: ((range_pointer(cv_base(pointer(uint))))):
               not_null?(pointer(uint))(first_value) ∧
                first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))
                ⊃ (∀ (i: below[N]): Cpp_Type?(uint)))
```

Repeatedly Skolemizing and flattening,

Using lemma Search_Pre_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `read_array_TCC2`.
Q.E.D.

C.198.19 Search_Example.read_array_TCC3

Terse proof for `read_array_TCC3`.

`read_array_TCC3`:

```
{1}  ∀ (s: State):
       read_first(s) ⊃
        (∀ (first_value: ((range_pointer(cv_base(pointer(uint))))):
             not_null?(pointer(uint))(first_value) ∧
              first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))
              ⊃
              (∀ (i: below[N]):
                   up?[Memory_Address]
                        (address_of(pointer(uint))
                                  (add(pointer(uint))
                                       (first_value, i × size_of(uint))))))
```

Repeatedly Skolemizing and flattening,
Rewriting using address_of_spec, matching in *,
Using lemma add_spec,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `read_array_TCC3`.
Q.E.D.

C.198.20 Search_Example.not_found_TCC1

Terse proof for `not_found_TCC1`.

`not_found_TCC1`:

```
{1}  ∀ (s: State):
       read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s) ⊃
        (∀ (first_value: ((range_pointer(cv_base(pointer(uint))))):
             first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
              (∀ (i: below[N]): Cpp_Type?(uint)))
```

Repeatedly Skolemizing and flattening,
Using lemma Search_Pre_TCC1,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `not_found_TCC1`.
Q.E.D.

C.198.21 Search_Example.not_found_TCC2

Terse proof for `not_found_TCC2`.

not_found_TCC2:

```
{1}   ∀ (s: State):
        read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s) ⊃
         (∀ (first_value: ((range_pointer(cv_base(pointer(uint))))))):
              first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
               (∀ (i: below[N]):
                    up?[Memory_Address]
                         (address_of(pointer(uint))
                                    (add(pointer(uint))
                                         (first_value, i × size_of(uint))))))
```

Repeatedly Skolemizing and flattening,
Rewriting using address_of_spec, matching in *,
Using lemma add_spec,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of read_array,
which is trivially true.
This completes the proof of not_found_TCC2.
Q.E.D.

C.198.22 Search_Example.not_found_TCC3

Terse proof for not_found_TCC3.

not_found_TCC3:

```
{1}   ∀ (s: State):
        read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s) ⊃
         (∀ (first_value: ((range_pointer(cv_base(pointer(uint))))))):
              first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s)) ⊃
               (∀ (i: below[N]):
                    OK?[State, ((range(uint)))]
                         (read_data[State, ((range(uint)))]
                                   (pm[State], dt(uint))
                                   (down[Memory_Address]
                                         (address_of(pointer(uint))
                                                    (add(pointer(uint))
                                                         (first_value, i × size_of(uint)))))
                                   (s))))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of read_array,
Replacing using formula -7,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
This completes the proof of not_found_TCC3.
Q.E.D.

C.198.23 Search_Example.not_found_TCC4

Terse proof for not_found_TCC4.

not_found_TCC4:

```
{1}  ∀ (s: State):
       read_first(s) ∧
        read_current(s) ∧
         read_last(s) ∧
          read_array(s) ∧
           (LET first_value = data(read_data(pm, dt_cv_pointer(pointer(uint)))(first)(s))
                IN
                ∀ (i: below[N]):
                  (data(read_data(pm, dt(uint))
                                  (down(address_of(pointer(uint))
                                                  (add(pointer(uint))
                                                      (first_value,
                                                       i × size_of(uint)))))
                                  (s))
                       ≠ value))
        ⊃
        OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
            (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
                 (pm[State], dt_cv_pointer(pointer(uint)))(last)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of read_last,
which is trivially true.
This completes the proof of not_found_TCC4.
Q.E.D.

C.198.24 Search_Example.found_TCC1

Terse proof for found_TCC1.

found_TCC1:

```
{1}  ∀ (s: State):
       deref_current(s) ∧
        read_array(s) ∧ read_last(s) ∧ read_current(s) ∧ read_first(s)
        ⊃
        OK?[State, ((range_pointer(cv_base(pointer(uint)))))]
            (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
                 (pm[State], dt_cv_pointer(pointer(uint)))(last)(s))
```

Repeatedly Skolemizing and flattening,
Expanding the definition of read_last,
which is trivially true.
This completes the proof of found_TCC1.
Q.E.D.

C.198.25 Search_Example.found_TCC2

Terse proof for found_TCC2.

`found_TCC2:`

```
{1}   ∀ (s: State):
        read_first(s) ∧
         read_current(s) ∧
          read_last(s) ∧
           read_array(s) ∧
            deref_current(s) ∧
             data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) ≠
              data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))
         ⊃
         up?[Memory_Address]
              (address_of(pointer(uint))
                          (data[State, ((range_pointer(cv_base(pointer(uint)))))]
                                (read_data[State, ((range_pointer(cv_base(pointer(uint)))))]
                                      (pm[State], dt_cv_pointer(pointer(uint)))(current
                                      (s))))
```

Repeatedly Skolemizing and flattening,

Expanding the definition of deref_current,

Applying disjunctive simplification to flatten sequent,

Rewriting using address_of_spec, matching in *,

Expanding the definition of cv_base,

which is trivially true.

This completes the proof of `found_TCC2`.

Q.E.D.

C.198.26 Search_Example.found_TCC3

Terse proof for `found_TCC3`.

`found_TCC3:`

```
{1}  ∀ (s: State):
       read_first(s) ∧
        read_current(s) ∧
         read_last(s) ∧
           read_array(s) ∧
            deref_current(s) ∧
             data(read_data(pm, dt_cv_pointer(pointer(uint)))(last)(s)) ≠
              data(read_data(pm, dt_cv_pointer(pointer(uint)))(current)(s))
       ⊃
       OK?[State, ((range(uint)))]
            (read_data[State, ((range(uint)))]
                  (pm[State], dt(uint))
                  (down[Memory_Address]
                        (address_of(pointer(uint))
                                    (data[State, ((range_pointer(cv_base(pointer(uint)))))]
                                          (read_data
                                                [State,
                                                  ((range_pointer(cv_base(pointer(uint)))))]
                                                (pm[State], dt_cv_pointer(pointer(uint)))
                                                (current)(s)))))
                  (s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `found_TCC3`.
Q.E.D.

C.198.27 Search_Example.uchar_range_values_TCC1

Terse proof for `uchar_range_values_TCC1`.

`uchar_range_values_TCC1:`

```
{1}  ∀ (n: nat): n ≤ 255 ⊃ Cpp_Type?(uchar) ∧ cv(non_bool_integral_enum?)(uchar)
```

Repeatedly Skolemizing and flattening,
Applying propositional simplification,
we get 2 subgoals:

`uchar_range_values_TCC1.1:`

```
{-1}  n′ ≥ 0
{-2}  n′ ≤ 255
-------------------------------------------------------------------------------
 {1}  Cpp_Type?(uchar)
```

Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array?

enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void? no_array_of_bitfields?

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `uchar_range_values_TCC1.1`.

`uchar_range_values_TCC1.2`:

{-1}	$n' \geq 0$
{-2}	$n' \leq 255$
{1}	cv(non_bool_integral_enum?)(uchar)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `uchar_range_values_TCC1.2`.

Q.E.D.

C.198.28 Search_Example.uchar_range_values

Terse proof for `uchar_range_values`.

`uchar_range_values`:

{1}	$\forall$ (n: nat): $n \leq 255 \supset$ range(uchar)(n)

Repeatedly Skolemizing and flattening,

Expanding the definition of range,

Expanding the definition of range_integral,

Expanding the definition of extend,

Using lemma binary_range_uchar,

Using lemma max_uchar,

Using lemma min_max_value_bits_uchar,

Trying repeated skolemization, instantiation, and if-lifting,

Using lemma expt_ge1,

Expanding the definition of ^,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `uchar_range_values`.

Q.E.D.

C.198.29 Search_Example.search_terminates

Terse proof for `search_terminates`.

`search_terminates`:

{1}	Valid(Search_Pre, search, λ (s: State): TRUE)

Installing automatic rewrites from: valid Valid Search_Pre search

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr

assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr

comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_terminates`.

Q.E.D.

C.198.30 Search_Example.search_read_first

Terse proof for `search_read_first`.

`search_read_first:`

{1} valid(Search_Pre, search, read_first)

Installing automatic rewrites from: valid Valid Search_Pre search read_first search_terminates

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_read_first`.
Q.E.D.

C.198.31 Search_Example.search_read_current

Terse proof for `search_read_current`.
`search_read_current:`

{1} valid(Search_Pre, search, read_current)

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current search_read_first search_terminates

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr

break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_read_current`.

Q.E.D.

C.198.32 Search_Example.search_deref_current

Terse proof for `search_deref_current`.

`search_deref_current:`

{1} valid(Search_Pre, search, deref_current)

Installing automatic rewrites from: valid Valid Search_Pre *N* search read_first deref_current search_read_current read_current search_terminates search_read_first

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 address_add_zero address_add_sum same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr

comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstmt_expr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Instantiating the top quantifier in -14 with the terms: (0),

Instantiating the top quantifier in -15 with the terms: (0),

Instantiating the top quantifier in -16 with the terms: (0),

Instantiating the top quantifier in -17 with the terms: (0),

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_deref_current`.

Q.E.D.

C.198.33 Search_Example.search_read_last

Terse proof for `search_read_last`.

`search_read_last:`

```
{1}  valid(Search_Pre, search, read_last)
```

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current search_read_first deref_current read_last search_current search_deref_current search_terminates *N*

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr

expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `search_read_last`.

Q.E.D.

C.198.34 Search_Example.search_read_array

Terse proof for `search_read_array`.

`search_read_array:`

{1} valid(Search_Pre, search, read_array)

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current search_read_first deref_current read_last search_current search_deref_current search_read_last read_array search_terminates *N*

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier in-

stantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `search_read_array`.

Q.E.D.

C.198.35 Search_Example.search_not_found

Terse proof for `search_not_found`.

`search_not_found:`

{1} valid(Search_Pre, search, not_found)

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current search_read_first deref_current read_last search_current search_deref_current search_read_last read_array search_read_array not_found search_terminates N

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default

return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok! stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `search_not_found`.

Q.E.D.

C.198.36 Search_Example.search_found

Terse proof for `search_found`.

`search_found:`

{1} valid(Search_Pre, search, found)

Installing automatic rewrites from: valid Valid Search_Pre search read_first read_current search_read_first deref_current read_last search_current search_deref_current search_read_last read_array

search_read_array not_found search_not_found found search_terminates N

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: mod_range size_of uidt er_plus_ax er_minus_ax er_div_ax er_times_ax er_neg_ax address_of_spec address_of_spec2 add_spec add_spec2 same_array_spec same_array_spec2 check_bounds_spec extended_real_superset_of_number_fields

Installing automatic rewrites from: literal id member postinc postdec deref arrow unary_minus unary_minus_unsigned unary_not unary_bitnot preinc predec UnaryExpressions.sizeof ptm_member ptm_arrow arrow times div div_float mod plus plus_ptr postinc_ptr preinc_ptr subscript minus minus_ptr postdec_ptr predec_ptr minus_ptr_ptr cmp lt gt le ge lt_ptr gt_ptr le_ptr ge_ptr cmp_pointer eq_bool eq not_equal not_equal_bool eq_ptr not_equal_ptr bitand bitxor bitor and_exp or_exp conditional_expr assign assign_times assign_div assign_mod assign_div_float assign_plus assign_plus_ptr assign_minus assign_minus_ptr assign_bitand assign_bitxor assign_bitor comma

Installing automatic rewrites from: plain_memory_write_ok_single plain_memory_read_data_ok plain_memory_write_ok_q_expr plain_memory_write_ok_q_stmt in_blessed_memory_rw_ro pm_q_prop_ok pm_q_prop_single_read pm_q_prop_single_write pm_q_prop_read_write_single pm_q_prop_read_write_other_single pm_q_prop_read_read_single plain_memory_read_write_other_single_data plain_memory_read_read plain_memory_read_write_res pm_q_prop_read_ok_expr pm_q_prop_read_ok_stmt pm_q_prop_read_pm_q_prop_expr pm_q_prop_read_pm_q_prop_stmt pm_q_prop_read_write_q_expr pm_q_prop_read_write_q_stmt plain_memory_read_write_q_data_expr plain_memory_read_write_q_data_stmt pm_q_prop_read_write_other_q_expr pm_q_prop_read_write_other_q_stmt pm_q_prop_read_read_q_expr pm_q_prop_read_read_q_stmt plain_memory_read_write_other_q_data_expr plain_memory_read_write_other_q_data_stmt plain_memory_read_read_q_data_exp plain_memory_read_read_q_data_stmt pm_q_prop_ok_result_single pm_q_prop_ok_result_q_expr pm_q_prop_ok_result_q_stmt pm_q_prop_read_ok_result_single pm_q_prop_read_ok_result_q_expr pm_q_prop_read_ok_result_q_stmt pm_q_prop_e2s pm_q_prop_stmt_e2s pm_q_prop_read_e2s pm_q_prop_read_stmt_e2s pm_q_prop_read_ok__pm_q_prop_read_expr pm_q_prop_read_ok__pm_q_prop_read_stmt

Installing automatic rewrites from: if_else switch member break_catch_break break_break_stmt break_break_catch_continue break_break_catch_default break_break_lift_stmt break_break_case break_break_default break_break stmt_break_break break_stmt_break break_stmt_catch_continue break_stmt_catch_default break_stmt_lift break_stmt_case break_stmt_default continue_catch_continue continue_break_stmt continue_break_catch_continue continue_break_catch_default continue_break_lift_stmt continue_break_case continue_break_default continue_continue stmt_continue_continue continue_stmt_continue continue_stmt_catch_continue continue_stmt_catch_default continue_stmt_lift continue_stmt_case continue_stmt_default switch_stmt stmt_switch_stmt switch_case_taken stmt_switch_case_taken switch_case_not_taken stmt_switch_case_not_taken switch_default stmt_switch_default switch_catch_break stmt_switch_catch_break switch_catch_continue stmt_switch_catch_continue switch_catch_default stmt_switch_catch_default default_stmt stmt_default_stmt default_case stmt_default_case default_default stmt_default_default default_catch_break stmt_default_catch_break default_catch_continue stmt_default_catch_continue default_catch_default stmt_default_catch_default return_void_result return_ex_result stmt_remove_catch_default stmt_remove_case stmt_remove_default stmt_remove_catch_continue stmt_remove_catch_break break_break_stmt_expr break_break_catch_continue_expr break_break_catch_default_expr break_break_lift_stmt_expr break_break_case_expr break_break_default_expr continue_break_stmt_expr continue_break_catch_continue_expr continue_break_catch_default_expr continue_break_lift_stmt_expr continue_break_case_expr continue_break_default_expr

Installing automatic rewrites from: skip_ok skip_state skip_elemination stmt_skip_elemination stmt_ok_lift stmt_ok_lift_expr e2s_ok stmt_e2s_ok e2s_state stmt_e2s_state ok_result_fexpr ok_result_fstmt e2s_merge stmt_e2s_merge e2s_expr stmt_e2s_expr

Installing automatic rewrites from: while_unroll stmt_while_unroll iterate_while_ok while_hang while_unroll_expr while_unroll_lexpr stmt_while_unroll_expr stmt_while_unroll_lexpr while_unroll_0 do_while_unroll do_while_inv_unroll for_unroll for_inv_unroll while_to_while_no_cb while_invariant?! while_variant?! while_inv_rewrite_termination_result! while_inv_rewrite_data_ok! while_inv_rewrite_data_break! while_inv_rewrite_data_return! stmt_while_inv_rewrite_termination_result! stmt_while_inv_rewrite_data_ok!

stmt_while_inv_rewrite_data_break! stmt_while_inv_rewrite_data_return! pm_q_prop_while! pm_q_prop_stmt_while!

Installing automatic rewrites from: composition_assoc_rewrite_stmt_ok forward_s_c_s_of_s_expr comp_eval_if_ok_fstmt stmt_eval_if_ok_fstmt comp_eval_if_ok_fstmt_cstmt stmt_eval_if_ok_fstmt_cstmt composition_assoc_rewrite_1 composition_assoc_rewrite_2 composition_assoc_rewrite_3 composition_assoc_rewrite_4 composition_assoc_rewrite_5 composition_assoc_rewrite_6 composition_assoc_rewrite_7 composition_assoc_rewrite_8 composition_assoc_rewrite_expr_1 composition_assoc_rewrite_expr_2 composition_assoc_rewrite_expr_3 composition_assoc_rewrite_expr_4 composition_assoc_rewrite_expr_5 composition_assoc_rewrite_expr_6 composition_assoc_rewrite_stmt_ok composition_assoc_rewrite_stmt_ok_expr composition_assoc_rewrite_stmt_ok_expr_lexpr comp_eval_if_ok_fstmt comp_eval_if_ok_fstmt_expr comp_eval_if_ok_fstmt_cstmt_expr stmt_eval_if_ok_fstmt_expr stmt_eval_if_ok_fstmt_cstr comp_eval_if_ok_fexpr_expr comp_eval_if_ok_fexpr stmt_eval_if_ok_fexpr expr_eval_if_ok_fexpr composition_assoc_expression_rewrite composition_assoc_expression_rewrite_stmt stmt_eval_if_ok_fexpr_expr expr_eval_if_ok_fexpr_expr ok_result_fexpr ok_result_fstmt composition_assoc_rewrite_stmt_ok_cstmt

Installing automatic rewrites from: allocate_stack deallocate_stack with_new_stackvar

Installing automatic rewrites from: ok_result_ok ok_result_data ok_result_q_ok ok_result_q_state ok_result_q_data ok_result_elemination_1 ok_result_elemination_2 ok_result_state

Installing automatic rewrites from: l2r integral_to_bool pointer_to_bool bool2int

Installing automatic rewrites from: comp_simple_expr_simple_expr_ok comp_simple_expr_simple_expr_data comp_simple_expr_simple_expr_state comp_simple_stmt_simple_expr_ok comp_simple_stmt_simple_expr_data comp_simple_stmt_simple_expr_state

Installing automatic rewrites from: floating_point?

Installing automatic rewrites from: cv_base_result2 cv_base_result_c cv_base_result_v cv_base_result_cv cv c v

Installing automatic rewrites from: uchar_range_values

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `search_found`.

Q.E.D.

C.199 Proofs for Segment_Reg_Datatype (paging-data.pvs)

C.199.1 Segment_Reg_Datatype.segment_selector_data_type_TCC1

Terse proof for `segment_selector_data_type_TCC1`.

`segment_selector_data_type_TCC1:`

```
{1}  ∃ (x: (pod_data_type?[Segment_Selector_type])): TRUE
```

Rewriting using segment_selector_type_exists, matching in *,

This completes the proof of `segment_selector_data_type_TCC1`.

Q.E.D.

C.199.2 Segment_Reg_Datatype.segment_reg_data_type_TCC1

Terse proof for `segment_reg_data_type_TCC1`.

`segment_reg_data_type_TCC1:`

```
{1}  ∃ (x: (pod_data_type?[Segment_Reg_type])): TRUE
```

Using lemma segment_reg_data_type_exists,
This completes the proof of `segment_reg_data_type_TCC1`.
Q.E.D.

C.200 Proofs for Segment_Types (paging-data.pvs)

This theory contains no provable formal statements.

C.201 Proofs for SelectionStatements (statements.pvs)

This theory contains no provable formal statements.

C.202 Proofs for Set_Lift (vfiasco-prelude.pvs)

This theory contains no provable formal statements.

C.203 Proofs for Single_Statement_Rewrites0 (statement-rewrites.pvs)

C.203.1 Single_Statement_Rewrites0.break_catch_break

Terse proof for `break_catch_break`.
`break_catch_break:`

{1} ∀ (s: State, stmt: [State → StmtResult[State]]):
OK?(stmt(s)) ⊃ (stmt ## break ## catch_break)(s) = stmt(s)

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `break_catch_break`.
Q.E.D.

C.203.2 Single_Statement_Rewrites0.break_break_stmt

Terse proof for `break_break_stmt`.
`break_break_stmt:`

{1} ∀ (s: State, stmt, stmt1: [State → StmtResult[State]]):
Break?(stmt(s)) ⊃ (stmt ## stmt1 ## catch_break)(s) = (stmt ## catch_break)(s)

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_stmt`.
Q.E.D.

C.203.3 Single_Statement_Rewrites0.break_break_catch_continue

Terse proof for `break_break_catch_continue`.

`break_break_catch_continue:`

$$\{1\}\quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset (\text{stmt} \mathbin{\#\#} \text{catch\_continue} \mathbin{\#\#} \text{catch\_break})(s) = (\text{stmt} \mathbin{\#\#} \text{catch\_break})(s)$$

,

This completes the proof of `break_break_catch_continue`.

Q.E.D.

C.203.4 Single_Statement_Rewrites0.break_break_catch_default

Terse proof for `break_break_catch_default`.

`break_break_catch_default:`

$$\{1\}\quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset (\text{stmt} \mathbin{\#\#} \text{catch\_default} \mathbin{\#\#} \text{catch\_break})(s) = (\text{stmt} \mathbin{\#\#} \text{catch\_break})(s)$$

,

This completes the proof of `break_break_catch_default`.

Q.E.D.

C.203.5 Single_Statement_Rewrites0.break_break_lift_stmt

Terse proof for `break_break_lift_stmt`.

`break_break_lift_stmt:`

$$\{1\}\quad \forall\ (s\colon \text{State},\ \text{stmt},\ \text{stmt1}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset (\text{stmt} \mathbin{\#\#} \text{lift}(\text{stmt1}) \mathbin{\#\#} \text{catch\_break})(s) = (\text{stmt} \mathbin{\#\#} \text{catch\_break})(s)$$

,

This completes the proof of `break_break_lift_stmt`.

Q.E.D.

C.203.6 Single_Statement_Rewrites0.break_break_case

Terse proof for `break_break_case`.

`break_break_case:`

$$\{1\}\quad \forall\ (n\colon \text{int},\ s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset (\text{stmt} \mathbin{\#\#} \text{case}(n) \mathbin{\#\#} \text{catch\_break})(s) = (\text{stmt} \mathbin{\#\#} \text{catch\_break})(s)$$

,

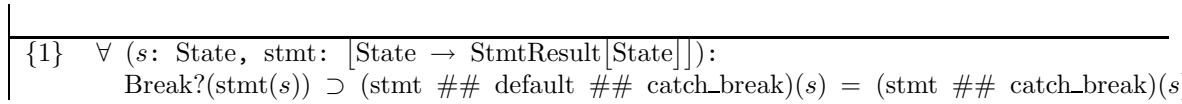
This completes the proof of `break_break_case`.

Q.E.D.

C.203.7 Single_Statement_Rewrites0.break_break_default

Terse proof for `break_break_default`.

`break_break_default:`



$\{1\}$ $\forall$ (s: State, stmt: $\lfloor$State $\to$ StmtResult$\lfloor$State$\rfloor\rfloor$):
Break?(stmt(s)) $\supset$ (stmt ## default ## catch_break)(s) = (stmt ## catch_break)(s)

,

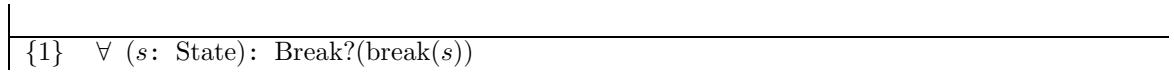
This completes the proof of `break_break_default`.

Q.E.D.

C.203.8 Single_Statement_Rewrites0.break_break

Terse proof for `break_break`.

`break_break:`



$\{1\}$ $\forall$ (s: State): Break?(break(s))

,

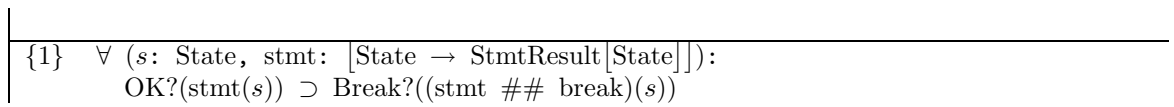
This completes the proof of `break_break`.

Q.E.D.

C.203.9 Single_Statement_Rewrites0.stmt_break_break

Terse proof for `stmt_break_break`.

`stmt_break_break:`



$\{1\}$ $\forall$ (s: State, stmt: $\lfloor$State $\to$ StmtResult$\lfloor$State$\rfloor\rfloor$):
OK?(stmt(s)) $\supset$ Break?((stmt ## break)(s))

,

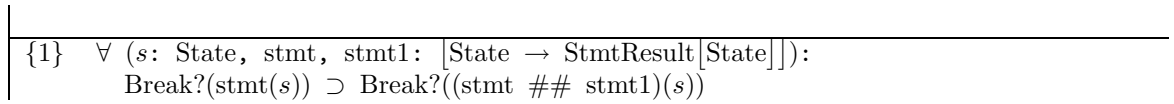
This completes the proof of `stmt_break_break`.

Q.E.D.

C.203.10 Single_Statement_Rewrites0.break_stmt_break

Terse proof for `break_stmt_break`.

`break_stmt_break:`



$\{1\}$ $\forall$ (s: State, stmt, stmt1: $\lfloor$State $\to$ StmtResult$\lfloor$State$\rfloor\rfloor$):
Break?(stmt(s)) $\supset$ Break?((stmt ## stmt1)(s))

,

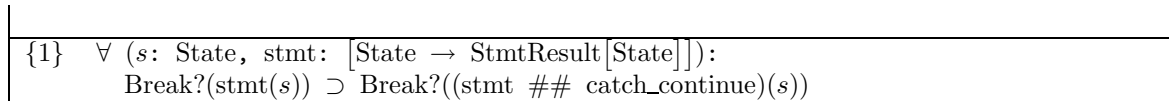
This completes the proof of `break_stmt_break`.

Q.E.D.

C.203.11 Single_Statement_Rewrites0.break_stmt_catch_continue

Terse proof for `break_stmt_catch_continue`.

`break_stmt_catch_continue:`



$\{1\}$ $\forall$ (s: State, stmt: $\lfloor$State $\to$ StmtResult$\lfloor$State$\rfloor\rfloor$):
Break?(stmt(s)) $\supset$ Break?((stmt ## catch_continue)(s))

,
This completes the proof of `break_stmt_catch_continue`.
Q.E.D.

C.203.12 Single_Statement_Rewrites0.break_stmt_catch_default

Terse proof for `break_stmt_catch_default`.
`break_stmt_catch_default:`

$$\{1\} \quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \mathbin{\#\#} \text{catch\_default})(s))$$

,
This completes the proof of `break_stmt_catch_default`.
Q.E.D.

C.203.13 Single_Statement_Rewrites0.break_stmt_lift

Terse proof for `break_stmt_lift`.
`break_stmt_lift:`

$$\{1\} \quad \forall\ (s\colon \text{State},\ \text{stmt},\ \text{stmt1}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \mathbin{\#\#} \text{lift}(\text{stmt1}))(s))$$

,
This completes the proof of `break_stmt_lift`.
Q.E.D.

C.203.14 Single_Statement_Rewrites0.break_stmt_case

Terse proof for `break_stmt_case`.
`break_stmt_case:`

$$\{1\} \quad \forall\ (n\colon \text{int},\ s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \mathbin{\#\#} \text{case}(n))(s))$$

,
This completes the proof of `break_stmt_case`.
Q.E.D.

C.203.15 Single_Statement_Rewrites0.break_stmt_default

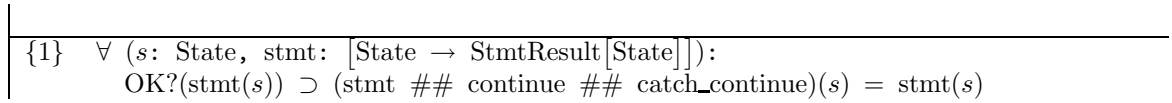
Terse proof for `break_stmt_default`.
`break_stmt_default:`

$$\{1\} \quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \text{Break?}(\text{stmt}(s)) \supset \text{Break?}((\text{stmt} \mathbin{\#\#} \text{default})(s))$$

,
This completes the proof of `break_stmt_default`.
Q.E.D.

C.203.16 Single_Statement_Rewrites0.continue_catch_continue

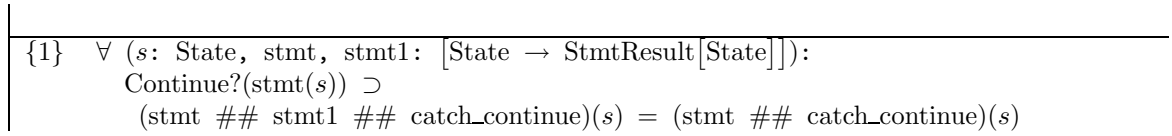
Terse proof for `continue_catch_continue`.
`continue_catch_continue:`



Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `continue_catch_continue`.
Q.E.D.

C.203.17 Single_Statement_Rewrites0.continue_break_stmt

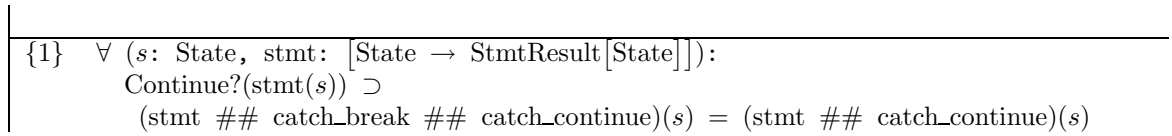
Terse proof for `continue_break_stmt`.
`continue_break_stmt:`



,
This completes the proof of `continue_break_stmt`.
Q.E.D.

C.203.18 Single_Statement_Rewrites0.continue_break_catch_continue

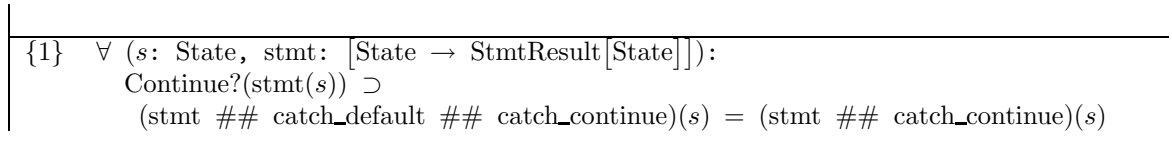
Terse proof for `continue_break_catch_continue`.
`continue_break_catch_continue:`



,
This completes the proof of `continue_break_catch_continue`.
Q.E.D.

C.203.19 Single_Statement_Rewrites0.continue_break_catch_default

Terse proof for `continue_break_catch_default`.
`continue_break_catch_default:`



,
This completes the proof of `continue_break_catch_default`.
Q.E.D.

C.203.20 Single_Statement_Rewrites0.continue_break_lift_stmt

Terse proof for `continue_break_lift_stmt`.

`continue_break_lift_stmt:`

```
{1}  ∀ (s: State, stmt, stmt1: [State → StmtResult[State]]):
       Continue?(stmt(s)) ⊃
         (stmt ## lift(stmt1) ## catch_continue)(s) = (stmt ## catch_continue)(s)
```

,

This completes the proof of `continue_break_lift_stmt`.

Q.E.D.

C.203.21 Single_Statement_Rewrites0.continue_break_case

Terse proof for `continue_break_case`.

`continue_break_case:`

```
{1}  ∀ (n: int, s: State, stmt: [State → StmtResult[State]]):
       Continue?(stmt(s)) ⊃
         (stmt ## case(n) ## catch_continue)(s) = (stmt ## catch_continue)(s)
```

,

This completes the proof of `continue_break_case`.

Q.E.D.

C.203.22 Single_Statement_Rewrites0.continue_break_default

Terse proof for `continue_break_default`.

`continue_break_default:`

```
{1}  ∀ (s: State, stmt: [State → StmtResult[State]]):
       Continue?(stmt(s)) ⊃
         (stmt ## default ## catch_continue)(s) = (stmt ## catch_continue)(s)
```

,

This completes the proof of `continue_break_default`.

Q.E.D.

C.203.23 Single_Statement_Rewrites0.continue_continue

Terse proof for `continue_continue`.

`continue_continue:`

```
{1}  ∀ (s: State): Continue?(continue(s))
```

,

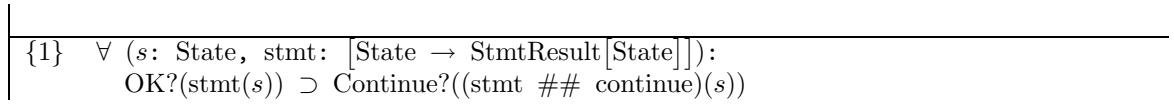
This completes the proof of `continue_continue`.

Q.E.D.

C.203.24 Single_Statement_Rewrites0.stmt_continue_continue

Terse proof for `stmt_continue_continue`.

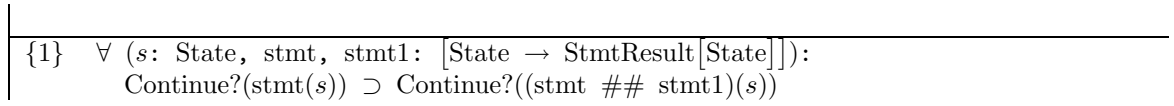
`stmt_continue_continue:`



,
This completes the proof of `stmt_continue_continue`.
Q.E.D.

C.203.25 Single_Statement_Rewrites0.continue_stmt_continue

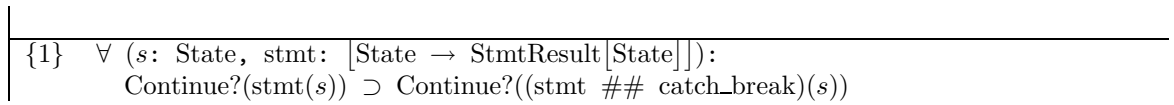
Terse proof for `continue_stmt_continue`.
`continue_stmt_continue:`



,
This completes the proof of `continue_stmt_continue`.
Q.E.D.

C.203.26 Single_Statement_Rewrites0.continue_stmt_catch_break

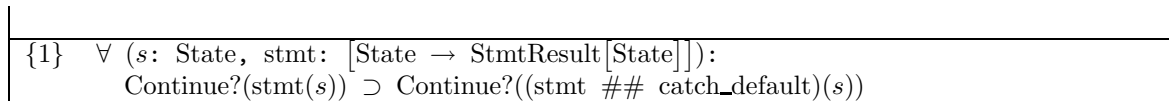
Terse proof for `continue_stmt_catch_break`.
`continue_stmt_catch_break:`



,
This completes the proof of `continue_stmt_catch_break`.
Q.E.D.

C.203.27 Single_Statement_Rewrites0.continue_stmt_catch_default

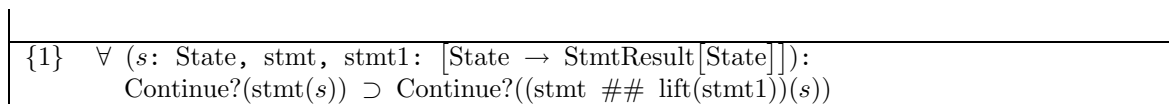
Terse proof for `continue_stmt_catch_default`.
`continue_stmt_catch_default:`



,
This completes the proof of `continue_stmt_catch_default`.
Q.E.D.

C.203.28 Single_Statement_Rewrites0.continue_stmt_lift

Terse proof for `continue_stmt_lift`.
`continue_stmt_lift:`



,

This completes the proof of `continue_stmt_lift`.
Q.E.D.

C.203.29 Single_Statement_Rewrites0.continue_stmt_case

Terse proof for `continue_stmt_case`.
`continue_stmt_case:`

$$\{1\} \quad \forall\ (n\colon \text{int},\ s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State} \rfloor \rfloor)\colon \\ \text{Continue?}(\text{stmt}(s)) \supset \text{Continue?}((\text{stmt} \mathbin{\#\#} \text{case}(n))(s))$$

,
This completes the proof of `continue_stmt_case`.
Q.E.D.

C.203.30 Single_Statement_Rewrites0.continue_stmt_default

Terse proof for `continue_stmt_default`.
`continue_stmt_default:`

$$\{1\} \quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State} \rfloor \rfloor)\colon \\ \text{Continue?}(\text{stmt}(s)) \supset \text{Continue?}((\text{stmt} \mathbin{\#\#} \text{default})(s))$$

,
This completes the proof of `continue_stmt_default`.
Q.E.D.

C.203.31 Single_Statement_Rewrites0.switch_stmt

Terse proof for `switch_stmt`.
`switch_stmt:`

$$\{1\} \quad \forall\ (n\colon \text{int},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State} \rfloor \rfloor)\colon \\ \text{switch\_result}(n) \mathbin{\#\#} \text{stmt} = \text{switch\_result}(n)$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `switch_stmt`.
Q.E.D.

C.203.32 Single_Statement_Rewrites0.stmt_switch_stmt

Terse proof for `stmt_switch_stmt`.
`stmt_switch_stmt:`

$$\{1\} \quad \forall\ (n\colon \text{int},\ \text{stmt},\ \text{stmt1}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State} \rfloor \rfloor)\colon \\ \text{stmt} \mathbin{\#\#} \text{switch\_result}(n) \mathbin{\#\#} \text{stmt1} = \text{stmt} \mathbin{\#\#} \text{switch\_result}(n)$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `stmt_switch_stmt`.
Q.E.D.

C.203.33 Single_Statement_Rewrites0.switch_case_taken

Terse proof for `switch_case_taken`.
`switch_case_taken:`

$$\{1\}\quad \forall\ (n\colon \text{int})\colon (\text{switch\_result}\lfloor\text{State}\rfloor(n)\ \#\#\ \text{case}(n)) = \text{skip}$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `switch_case_taken`.
Q.E.D.

C.203.34 Single_Statement_Rewrites0.stmt_switch_case_taken

Terse proof for `stmt_switch_case_taken`.
`stmt_switch_case_taken:`

$$\{1\}\quad \forall\ (n\colon \text{int},\ s\colon \text{State},\ \text{stmt}\colon \lfloor\text{State} \to \text{StmtResult}\lfloor\text{State}\rfloor\rfloor)\colon \\ \text{OK?}(\text{stmt}(s)) \supset (\text{stmt}\ \#\#\ \text{switch\_result}(n)\ \#\#\ \text{case}(n))(s) = \text{stmt}(s)$$

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `stmt_switch_case_taken`.
Q.E.D.

C.203.35 Single_Statement_Rewrites0.switch_case_not_taken

Terse proof for `switch_case_not_taken`.
`switch_case_not_taken:`

$$\{1\}\quad \forall\ (n_1,\ n_2\colon \text{int})\colon n_1 \neq n_2 \supset \text{switch\_result}\lfloor\text{State}\rfloor(n_1)\ \#\#\ \text{case}(n_2) = \text{switch\_result}(n_1)$$

Expanding the definition of ##,
Expanding the definition of case,
Expanding the definition of switch_result,
Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `switch_case_not_taken`.
Q.E.D.

C.203.36 Single_Statement_Rewrites0.stmt_switch_case_not_taken

Terse proof for `stmt_switch_case_not_taken`.

`stmt_switch_case_not_taken:`

$$\{1\}\quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor,\ n_1,\ n_2\colon \text{int})\colon \\ \text{OK?}(\text{stmt}(s)) \wedge n_1 \neq n_2 \supset \\ (\text{stmt} \mathbin{\#\#} \text{switch\_result}(n_1) \mathbin{\#\#} \text{case}(n_2))(s) = (\text{stmt} \mathbin{\#\#} \text{switch\_result}(n_1))(s)$$

,
This completes the proof of `stmt_switch_case_not_taken`.
Q.E.D.

C.203.37 Single_Statement_Rewrites0.switch_default

Terse proof for `switch_default`.
`switch_default:`

$$\{1\}\quad \forall\ (n\colon \text{int})\colon \text{switch\_result}\lfloor \text{State}\rfloor(n) \mathbin{\#\#} \text{default} = \text{switch\_result}(n)$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `switch_default`.
Q.E.D.

C.203.38 Single_Statement_Rewrites0.stmt_switch_default

Terse proof for `stmt_switch_default`.
`stmt_switch_default:`

$$\{1\}\quad \forall\ (n\colon \text{int},\ s\colon \text{State},\ \text{stmt}\colon \lfloor \text{State} \to \text{StmtResult}\lfloor \text{State}\rfloor\rfloor)\colon \\ \text{OK?}(\text{stmt}(s)) \supset \\ (\text{stmt} \mathbin{\#\#} \text{switch\_result}(n) \mathbin{\#\#} \text{default})(s) = (\text{stmt} \mathbin{\#\#} \text{switch\_result}(n))(s)$$

,
This completes the proof of `stmt_switch_default`.
Q.E.D.

C.203.39 Single_Statement_Rewrites0.switch_catch_break

Terse proof for `switch_catch_break`.
`switch_catch_break:`

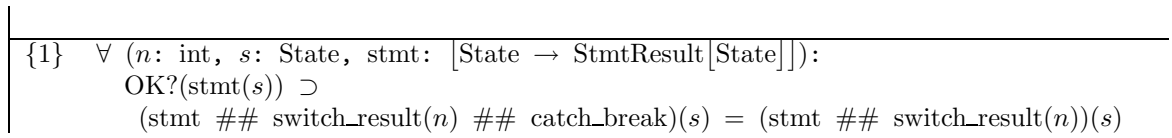
$$\{1\}\quad \forall\ (n\colon \text{int})\colon \text{switch\_result}\lfloor \text{State}\rfloor(n) \mathbin{\#\#} \text{catch\_break} = \text{switch\_result}(n)$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `switch_catch_break`.
Q.E.D.

C.203.40 Single_Statement_Rewrites0.stmt_switch_catch_break

Terse proof for `stmt_switch_catch_break`.

`stmt_switch_catch_break:`



,

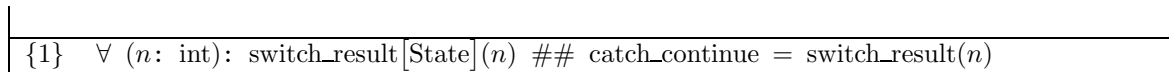
This completes the proof of `stmt_switch_catch_break`.

Q.E.D.

C.203.41 Single_Statement_Rewrites0.switch_catch_continue

Terse proof for `switch_catch_continue`.

`switch_catch_continue:`



Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

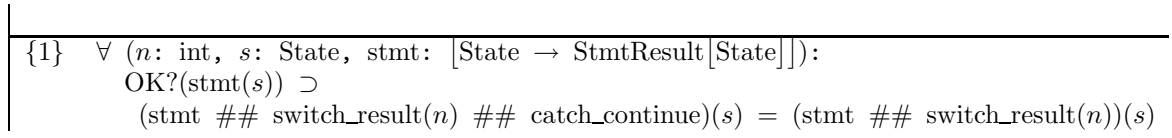
This completes the proof of `switch_catch_continue`.

Q.E.D.

C.203.42 Single_Statement_Rewrites0.stmt_switch_catch_continue

Terse proof for `stmt_switch_catch_continue`.

`stmt_switch_catch_continue:`



,

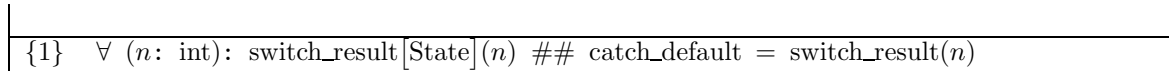
This completes the proof of `stmt_switch_catch_continue`.

Q.E.D.

C.203.43 Single_Statement_Rewrites0.switch_catch_default

Terse proof for `switch_catch_default`.

`switch_catch_default:`



Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `switch_catch_default`.

Q.E.D.

C.203.44 Single_Statement_Rewrites0.stmt_switch_catch_default

Terse proof for `stmt_switch_catch_default`.
`stmt_switch_catch_default:`

{1} ∀ (n: int, s: State, stmt: ⌈State → StmtResult⌈State⌉⌉):
OK?(stmt(s)) ⊃
(stmt ## switch_result(n) ## catch_default)(s) = (stmt ## switch_result(n))(s)

,
This completes the proof of `stmt_switch_catch_default`.
Q.E.D.

C.203.45 Single_Statement_Rewrites0.default_stmt

Terse proof for `default_stmt`.
`default_stmt:`

{1} ∀ (stmt: ⌈State → StmtResult⌈State⌉⌉): default_result ## stmt = default_result

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_stmt`.
Q.E.D.

C.203.46 Single_Statement_Rewrites0.stmt_default_stmt

Terse proof for `stmt_default_stmt`.
`stmt_default_stmt:`

{1} ∀ (stmt, stmt1: ⌈State → StmtResult⌈State⌉⌉):
stmt ## default_result ## stmt1 = stmt ## default_result

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_default_stmt`.
Q.E.D.

C.203.47 Single_Statement_Rewrites0.default_case

Terse proof for `default_case`.
`default_case:`

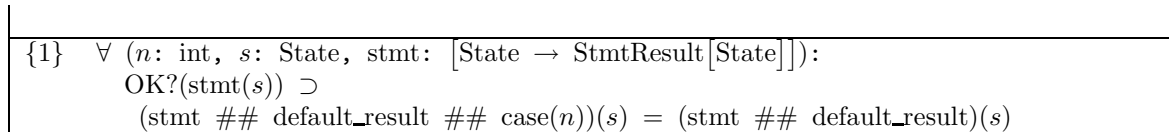
{1} ∀ (n: int): default_result⌈State⌉ ## case(n) = default_result

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_case`.
Q.E.D.

C.203.48 Single_Statement_Rewrites0.stmt_default_case

Terse proof for `stmt_default_case`.

`stmt_default_case:`



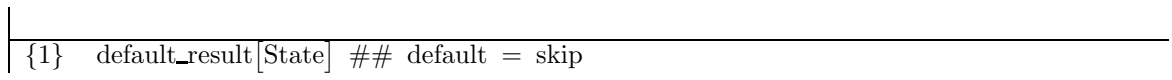
$\{1\}$ $\forall$ (n: int, s: State, stmt: $\lfloor$State $\to$ StmtResult$\lfloor$State$\rfloor\rfloor$):
OK?(stmt(s)) $\supset$
(stmt ## default_result ## case(n))(s) = (stmt ## default_result)(s)

,
This completes the proof of `stmt_default_case`.
Q.E.D.

C.203.49 Single_Statement_Rewrites0.default_default

Terse proof for `default_default`.

`default_default:`



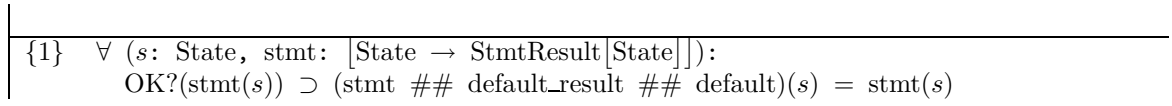
$\{1\}$ default_result$\lfloor$State$\rfloor$ ## default = skip

Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_default`.
Q.E.D.

C.203.50 Single_Statement_Rewrites0.stmt_default_default

Terse proof for `stmt_default_default`.

`stmt_default_default:`



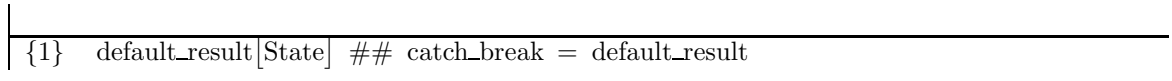
$\{1\}$ $\forall$ (s: State, stmt: $\lfloor$State $\to$ StmtResult$\lfloor$State$\rfloor\rfloor$):
OK?(stmt(s)) $\supset$ (stmt ## default_result ## default)(s) = stmt(s)

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `stmt_default_default`.
Q.E.D.

C.203.51 Single_Statement_Rewrites0.default_catch_break

Terse proof for `default_catch_break`.

`default_catch_break:`



$\{1\}$ default_result$\lfloor$State$\rfloor$ ## catch_break = default_result

Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_catch_break`.
Q.E.D.

C.203.52 Single_Statement_Rewrites0.stmt_default_catch_break

Terse proof for `stmt_default_catch_break`.
`stmt_default_catch_break:`

{1} ∀ (s: State, stmt: [State → StmtResult[State]]):
OK?(stmt(s)) ⊃
(stmt ## default_result ## catch_break)(s) = (stmt ## default_result)(s)

,
This completes the proof of `stmt_default_catch_break`.
Q.E.D.

C.203.53 Single_Statement_Rewrites0.default_catch_continue

Terse proof for `default_catch_continue`.
`default_catch_continue:`

{1} default_result[State] ## catch_continue = default_result

Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_catch_continue`.
Q.E.D.

C.203.54 Single_Statement_Rewrites0.stmt_default_catch_continue

Terse proof for `stmt_default_catch_continue`.
`stmt_default_catch_continue:`

{1} ∀ (s: State, stmt: [State → StmtResult[State]]):
OK?(stmt(s)) ⊃
(stmt ## default_result ## catch_continue)(s) = (stmt ## default_result)(s)

,
This completes the proof of `stmt_default_catch_continue`.
Q.E.D.

C.203.55 Single_Statement_Rewrites0.default_catch_default

Terse proof for `default_catch_default`.
`default_catch_default:`

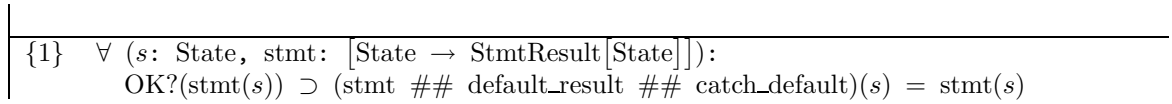
{1} default_result[State] ## catch_default = skip

Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `default_catch_default`.
Q.E.D.

C.203.56 Single_Statement_Rewrites0.stmt_default_catch_default

Terse proof for `stmt_default_catch_default`.

`stmt_default_catch_default:`

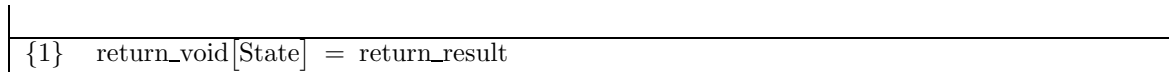


Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `stmt_default_catch_default`.
Q.E.D.

C.203.57 Single_Statement_Rewrites0.return_void_result

Terse proof for `return_void_result`.

`return_void_result:`

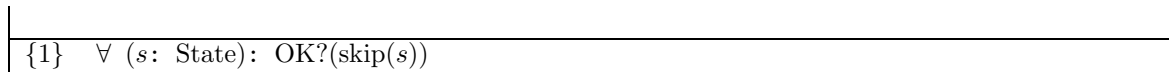


,
This completes the proof of `return_void_result`.
Q.E.D.

C.203.58 Single_Statement_Rewrites0.skip_ok

Terse proof for `skip_ok`.

`skip_ok:`

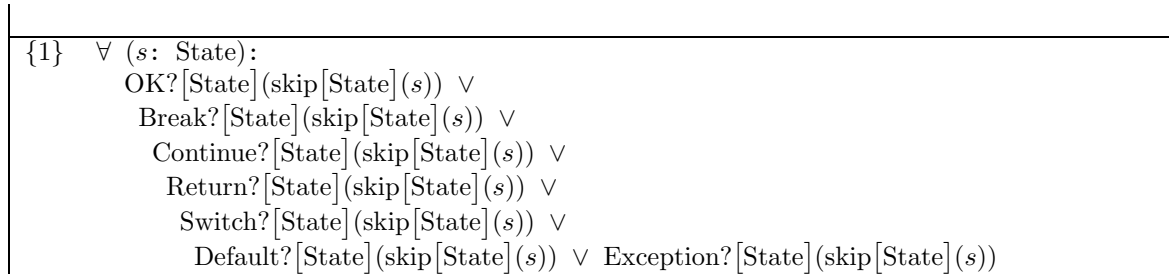


,
This completes the proof of `skip_ok`.
Q.E.D.

C.203.59 Single_Statement_Rewrites0.skip_state_TCC1

Terse proof for `skip_state_TCC1`.

`skip_state_TCC1:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `skip_state_TCC1`.
Q.E.D.

C.203.60 Single_Statement_Rewrites0.skip_state

Terse proof for `skip_state`.
`skip_state:`

$$\{1\} \quad \forall\ (s\colon \text{State})\colon \text{state}(\text{skip}(s)) = s$$

,
This completes the proof of `skip_state`.
Q.E.D.

C.203.61 Single_Statement_Rewrites0.skip_elemination

Terse proof for `skip_elemination`.
`skip_elemination:`

$$\{1\} \quad \forall\ (\text{stmt}\colon \lceil \text{State} \to \text{StmtResult}\lceil \text{State} \rceil \rceil)\colon (\text{stmt} \mathbin{\#\#} \text{skip}) = \text{stmt}$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
Applying extensionality,
This completes the proof of `skip_elemination`.
Q.E.D.

C.203.62 Single_Statement_Rewrites0.stmt_skip_elemination

Terse proof for `stmt_skip_elemination`.
`stmt_skip_elemination:`

$$\{1\} \quad \forall\ (\text{stmt}\colon \lceil \text{State} \to \text{StmtResult}\lceil \text{State} \rceil \rceil)\colon (\text{skip} \mathbin{\#\#} \text{stmt}) = \text{stmt}$$

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_skip_elemination`.
Q.E.D.

C.203.63 Single_Statement_Rewrites0.stmt_remove_catch_default

Terse proof for `stmt_remove_catch_default`.
`stmt_remove_catch_default:`

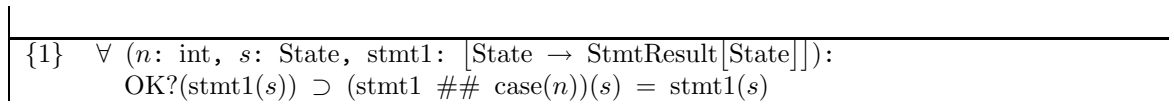
$$\{1\} \quad \forall\ (s\colon \text{State},\ \text{stmt}\colon \lceil \text{State} \to \text{StmtResult}\lceil \text{State} \rceil \rceil)\colon \\ \quad \text{OK?}(\text{stmt}(s)) \supset (\text{stmt} \mathbin{\#\#} \text{catch\_default})(s) = \text{stmt}(s)$$

,
This completes the proof of `stmt_remove_catch_default`.
Q.E.D.

C.203.64 Single_Statement_Rewrites0.stmt_remove_case

Terse proof for `stmt_remove_case`.

`stmt_remove_case:`



,

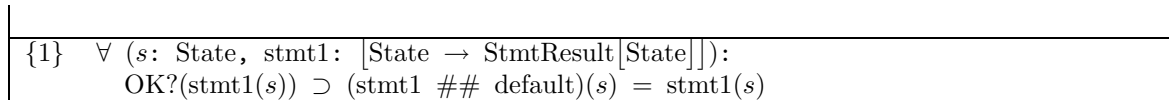
This completes the proof of `stmt_remove_case`.

Q.E.D.

C.203.65 Single_Statement_Rewrites0.stmt_remove_default

Terse proof for `stmt_remove_default`.

`stmt_remove_default:`



,

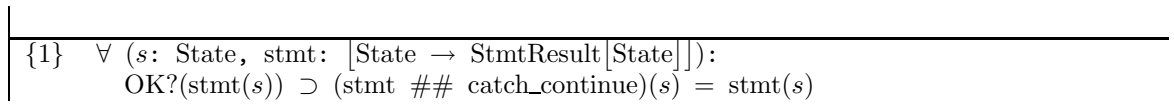
This completes the proof of `stmt_remove_default`.

Q.E.D.

C.203.66 Single_Statement_Rewrites0.stmt_remove_catch_continue

Terse proof for `stmt_remove_catch_continue`.

`stmt_remove_catch_continue:`



,

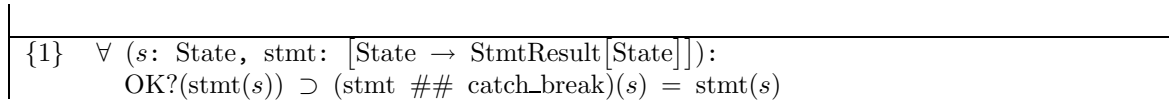
This completes the proof of `stmt_remove_catch_continue`.

Q.E.D.

C.203.67 Single_Statement_Rewrites0.stmt_remove_catch_break

Terse proof for `stmt_remove_catch_break`.

`stmt_remove_catch_break:`



,

This completes the proof of `stmt_remove_catch_break`.

Q.E.D.

C.203.68 Single_Statement_Rewrites0.stmt_ok_lift

Terse proof for `stmt_ok_lift`.

`stmt_ok_lift`:

{1} ∀ (s: State, stmt1, stmt2: [State → StmtResult[State]]):
OK?(stmt1(s)) ⊃ (stmt1 ## lift(stmt2))(s) = (stmt1 ## stmt2)(s)

,

This completes the proof of `stmt_ok_lift`.

Q.E.D.

C.203.69 Single_Statement_Rewrites0.while_unroll_TCC1

Terse proof for `while_unroll_TCC1`.

`while_unroll_TCC1`:

{1} ∀ (max: nat): max > 0 ⊃ max − 1 ≥ 0

Repeatedly Skolemizing and flattening,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `while_unroll_TCC1`.

Q.E.D.

C.203.70 Single_Statement_Rewrites0.while_unroll

Terse proof for `while_unroll`.

`while_unroll`:

{1} ∀ (b_ex: [State → ExprResult[State, bool]], max: nat, s: State,
stmt: [State → StmtResult[State]]):
max > 0 ⊃
while(max, b_ex, stmt)(s) =
if_else(b_ex, stmt ## catch_continue ## while(max − 1, b_ex, stmt) ## catch_break,
skip)
(s)

Repeatedly Skolemizing and flattening,

Expanding the definition of while,

Using lemma while_as_if_while,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `while_unroll`.

Q.E.D.

C.203.71 Single_Statement_Rewrites0.stmt_while_unroll

Terse proof for `stmt_while_unroll`.

`stmt_while_unroll:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]], max: nat, s: State,
stmt1, stmt2: [State → StmtResult[State]]):
max > 0 ⊃
(stmt1 ## while(max, b_ex, stmt2))(s) =
(stmt1 ##
if_else(b_ex,
stmt2 ## catch_continue ## while(max − 1, b_ex, stmt2) ## catch_break,
skip))
(s)

Repeatedly Skolemizing and flattening,

Expanding the definition of ##,

Expanding the definition of lift,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Rewriting using while_unroll, matching in *,

This completes the proof of `stmt_while_unroll`.

Q.E.D.

C.203.72 Single_Statement_Rewrites0.iterate_while_ok

Terse proof for `iterate_while_ok`.

`iterate_while_ok:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]], s_1: State,
stmt: [State → StmtResult[State]], n: nat):
(∀ (s: State): OK?((e2s(b_ex) ## stmt)(s))) ⊃
OK?(iterate_while(n, b_ex, stmt)(s_1))

Inducting on n on formula 1,

we get 2 subgoals:

`iterate_while_ok.1:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]], s_1: State,
stmt: [State → StmtResult[State]]):
(∀ (s: State): OK?((e2s(b_ex) ## stmt)(s))) ⊃
OK?(iterate_while(0, b_ex, stmt)(s_1))

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `iterate_while_ok.1`.

`iterate_while_ok.2:`

{1} ∀ j:
(∀ (b_ex: [State → ExprResult[State, bool]], s_1: State,
stmt: [State → StmtResult[State]]):
(∀ (s: State): OK?((e2s(b_ex) ## stmt)(s))) ⊃
OK?(iterate_while(j, b_ex, stmt)(s_1)))
⊃
(∀ (b_ex: [State → ExprResult[State, bool]], s_1: State,
stmt: [State → StmtResult[State]]):
(∀ (s: State): OK?((e2s(b_ex) ## stmt)(s))) ⊃
OK?(iterate_while($j+1$, b_ex, stmt)(s_1)))

Repeatedly Skolemizing and flattening,
Expanding the definition of iterate_while,
Instantiating quantified variables,
Expanding the definition of ##,
Expanding the definition of lift,
Case splitting on OK?(catch_continue((e2s(b_ex!1) ## stmt!1)(s1!1))),
we get 2 subgoals:
`iterate_while_ok.2.1:`

{-1} OK?(catch_continue((e2s(b_ex′) ## stmt′)(s_1')))
{-2} $j' \geq 0$
{-3} ∀ (s_1: State):
(∀ (s: State): OK?((e2s(b_ex′) ## stmt′)(s))) ⊃
OK?(iterate_while(j', b_ex′, stmt′)(s_1))
{-4} ∀ (s: State): OK?((e2s(b_ex′) ## stmt′)(s))

{1} OK?(CASES catch_continue((e2s(b_ex′) ## stmt′)(s_1')) OF
OK(state): iterate_while(j', b_ex′, stmt′)(state)
ELSE catch_continue((e2s(b_ex′) ## stmt′)(s_1'))
ENDCASES)

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: (state(catch_continue((e2s(b_ex!1) ## stmt!1)(s1!1)))),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `iterate_while_ok.2.1`.
`iterate_while_ok.2.2:`

{-1} $j' \geq 0$
{-2} ∀ (s_1: State):
(∀ (s: State): OK?((e2s(b_ex′) ## stmt′)(s))) ⊃
OK?(iterate_while(j', b_ex′, stmt′)(s_1))
{-3} ∀ (s: State): OK?((e2s(b_ex′) ## stmt′)(s))

{1} OK?(catch_continue((e2s(b_ex′) ## stmt′)(s_1')))
{2} OK?(CASES catch_continue((e2s(b_ex′) ## stmt′)(s_1')) OF
OK(state): iterate_while(j', b_ex′, stmt′)(state)
ELSE catch_continue((e2s(b_ex′) ## stmt′)(s_1'))
ENDCASES)

Hiding formulas: 2,
Expanding the definition of catch_continue,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `iterate_while_ok.2.2`.
Q.E.D.

C.203.73 Single_Statement_Rewrites0.while_hang

Terse proof for `while_hang`.
`while_hang:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], stmt: [State → StmtResult[State]]):
       (∀ (s: State):
          OK?(b_ex(s)) ∧ data(b_ex(s)) ∧ OK?((e2s(b_ex) ## stmt)(s)))
        ⊃ while(b_ex, stmt) = hang_result
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Expanding the definition of while,
Expanding the definition of hang_result,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Repeatedly Skolemizing and flattening,
Expanding the definition of while_termination_point?,
Hiding formulas: 1,
Case splitting on OK?(iterate_while(n!1, b_ex!1, stmt!1)(x!1)),
we get 2 subgoals:
`while_hang.1:`

```
{-1}  OK?(iterate_while(n′, b_ex′, stmt′)(x′))
{-2}  n′ ≥ 0
{-3}  ¬ OK?(iterate_while(n′, b_ex′, stmt′)(x′)) ∨
       ¬ OK?(b_ex′(state(iterate_while(n′, b_ex′, stmt′)(x′)))) ∨
        ¬ data(b_ex′(state(iterate_while(n′, b_ex′, stmt′)(x′))))
{-4}  ∀ (s: State):
       OK?(b_ex′(s)) ∧ data(b_ex′(s)) ∧ OK?((e2s(b_ex′) ## stmt′)(s))
```

Instantiating the top quantifier in -4 with the terms: (state(iterate_while(n!1, b_ex!1, stmt!1)(x!1))),
we get 2 subgoals:
`while_hang.1.1:`

```
{-1}  OK?(iterate_while(n′, b_ex′, stmt′)(x′))
{-2}  n′ ≥ 0
{-3}  ¬ OK?(iterate_while(n′, b_ex′, stmt′)(x′)) ∨
       ¬ OK?(b_ex′(state(iterate_while(n′, b_ex′, stmt′)(x′)))) ∨
        ¬ data(b_ex′(state(iterate_while(n′, b_ex′, stmt′)(x′))))
{-4}  OK?(b_ex′(state(iterate_while(n′, b_ex′, stmt′)(x′)))) ∧
       data(b_ex′(state(iterate_while(n′, b_ex′, stmt′)(x′)))) ∧
        OK?((e2s(b_ex′) ## stmt′)(state(iterate_while(n′, b_ex′, stmt′)(x′))))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `while_hang.1.1`.

`while_hang.1.2:`

{-1} OK?(iterate_while(n', b_ex$'$, stmt$'$)(x'))
{-2} $n' \geq 0$
{-3} $\neg$ OK?(iterate_while(n', b_ex$'$, stmt$'$)(x')) $\vee$
$\neg$ OK?(b_ex$'$(state(iterate_while(n', b_ex$'$, stmt$'$)(x')))) $\vee$
$\neg$ data(b_ex$'$(state(iterate_while(n', b_ex$'$, stmt$'$)(x'))))

{1} OK?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x')) $\vee$
Break?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x')) $\vee$
Continue?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x')) $\vee$
Return?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x')) $\vee$
Switch?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x')) $\vee$
Default?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x')) $\vee$
Exception?[State](iterate_while[State](n', b_ex$'$, stmt$'$)(x'))

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `while_hang.1.2`.
`while_hang.2:`

{-1} $n' \geq 0$
{-2} $\neg$ OK?(iterate_while(n', b_ex$'$, stmt$'$)(x')) $\vee$
$\neg$ OK?(b_ex$'$(state(iterate_while(n', b_ex$'$, stmt$'$)(x')))) $\vee$
$\neg$ data(b_ex$'$(state(iterate_while(n', b_ex$'$, stmt$'$)(x'))))
{-3} $\forall$ (s: State):
OK?(b_ex$'$(s)) $\wedge$ data(b_ex$'$(s)) $\wedge$ OK?((e2s(b_ex$'$) ## stmt$'$)(s))

{1} OK?(iterate_while(n', b_ex$'$, stmt$'$)(x'))

Rewriting using iterate_while_ok, matching in *,
Keeping (-3 1) and hiding *,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `while_hang.2`.
Q.E.D.

C.203.74 Single_Statement_Rewrites0.composition_assoc_rewrite_stmt_ok

Terse proof for `composition_assoc_rewrite_stmt_ok`.
`composition_assoc_rewrite_stmt_ok:`

{1} $\forall$ (cstmt: [StmtResult[State] $\to$ StmtResult[State]], s: State,
stmt, stmt1: [State $\to$ StmtResult[State]]):
OK?(stmt(s)) $\supset$ (stmt ## (stmt1 ## cstmt))(s) = ((stmt ## stmt1) ## cstmt)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_stmt_ok`.
Q.E.D.

C.203.75 Single_Statement_Rewrites0.composition_assoc_rewrite_stmt_ok_cstmt

Terse proof for `composition_assoc_rewrite_stmt_ok_cstmt`.
`composition_assoc_rewrite_stmt_ok_cstmt`:

{1} ∀ (cstmt, cstmt1: [StmtResult[State] → StmtResult[State]], s: State,
stmt, stmt1: [State → StmtResult[State]]):
OK?(stmt(s)) ⊃
((stmt ## (stmt1 ## cstmt)) ## cstmt1)(s) =
(((stmt ## stmt1) ## cstmt) ## cstmt1)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_stmt_ok_cstmt`.
Q.E.D.

C.204 Proofs for Single_Statement_Rewrites1 (statement-rewrites.pvs)

C.204.1 Single_Statement_Rewrites1.forward_s_c_s_of_s_expr

Terse proof for `forward_s_c_s_of_s_expr`.
`forward_s_c_s_of_s_expr`:

{1} ∀ (cstmt: [StmtResult[State] → StmtResult[State]],
expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
(stmt ## cstmt)(s) = stmt(s) ⊃ (stmt ## cstmt ## expr)(s) = (stmt ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `forward_s_c_s_of_s_expr`.
Q.E.D.

C.204.2 Single_Statement_Rewrites1.stmt_ok_lift_expr

Terse proof for `stmt_ok_lift_expr`.
`stmt_ok_lift_expr`:

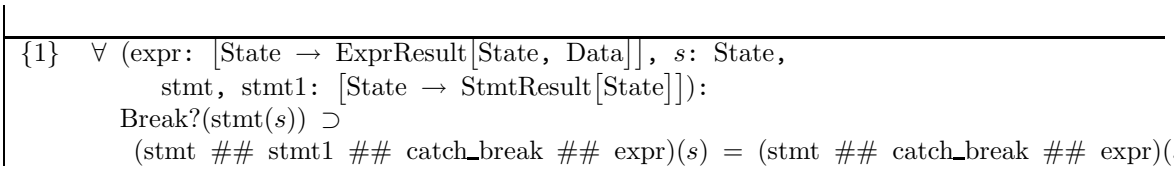
{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt1, stmt2: [State → StmtResult[State]]):
OK?(stmt1(s)) ⊃ (stmt1 ## lift(stmt2) ## expr)(s) = (stmt1 ## stmt2 ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_ok_lift_expr`.
Q.E.D.

C.204.3 Single_Statement_Rewrites1.break_break_stmt_expr

Terse proof for `break_break_stmt_expr`.

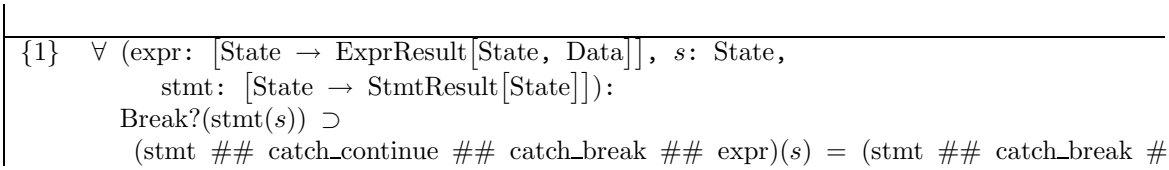
break_break_stmt_expr:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of break_break_stmt_expr.
Q.E.D.

C.204.4 Single_Statement_Rewrites1.break_break_catch_continue_expr

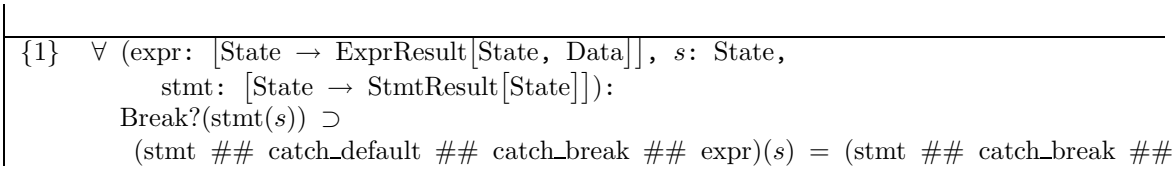
Terse proof for break_break_catch_continue_expr.
break_break_catch_continue_expr:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of break_break_catch_continue_expr.
Q.E.D.

C.204.5 Single_Statement_Rewrites1.break_break_catch_default_expr

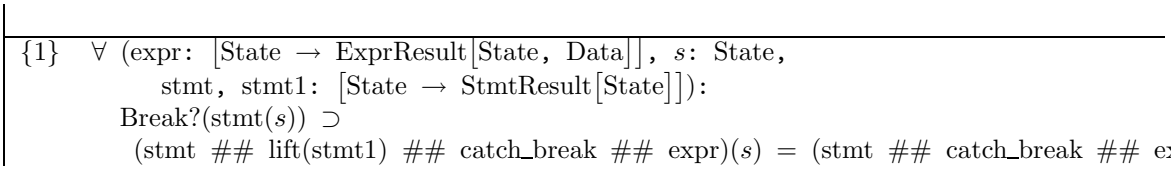
Terse proof for break_break_catch_default_expr.
break_break_catch_default_expr:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of break_break_catch_default_expr.
Q.E.D.

C.204.6 Single_Statement_Rewrites1.break_break_lift_stmt_expr

Terse proof for break_break_lift_stmt_expr.
break_break_lift_stmt_expr:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_lift_stmt_expr`.
Q.E.D.

C.204.7 Single_Statement_Rewrites1.break_break_case_expr

Terse proof for `break_break_case_expr`.
`break_break_case_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data]], n: int, s: State,
stmt: [State → StmtResult[State]]):
Break?(stmt(s)) ⊃
(stmt ## case(n) ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_case_expr`.
Q.E.D.

C.204.8 Single_Statement_Rewrites1.break_break_default_expr

Terse proof for `break_break_default_expr`.
`break_break_default_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
Break?(stmt(s)) ⊃
(stmt ## default ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `break_break_default_expr`.
Q.E.D.

C.204.9 Single_Statement_Rewrites1.continue_break_stmt_expr

Terse proof for `continue_break_stmt_expr`.
`continue_break_stmt_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt, stmt1: [State → StmtResult[State]]):
Continue?(stmt(s)) ⊃
(stmt ## stmt1 ## catch_break ## expr)(s) = (stmt ## catch_break ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_stmt_expr`.
Q.E.D.

C.204.10 Single_Statement_Rewrites1.continue_break_catch_continue_expr

Terse proof for `continue_break_catch_continue_expr`.

`continue_break_catch_continue_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
Continue?(stmt(s)) ⊃
(stmt ## catch_break ## catch_continue ## expr)(s) =
(stmt ## catch_continue ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_catch_continue_expr`.
Q.E.D.

C.204.11 Single_Statement_Rewrites1.continue_break_catch_default_expr

Terse proof for `continue_break_catch_default_expr`.

`continue_break_catch_default_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
Continue?(stmt(s)) ⊃
(stmt ## catch_default ## catch_continue ## expr)(s) =
(stmt ## catch_continue ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_catch_default_expr`.
Q.E.D.

C.204.12 Single_Statement_Rewrites1.continue_break_lift_stmt_expr

Terse proof for `continue_break_lift_stmt_expr`.

`continue_break_lift_stmt_expr:`

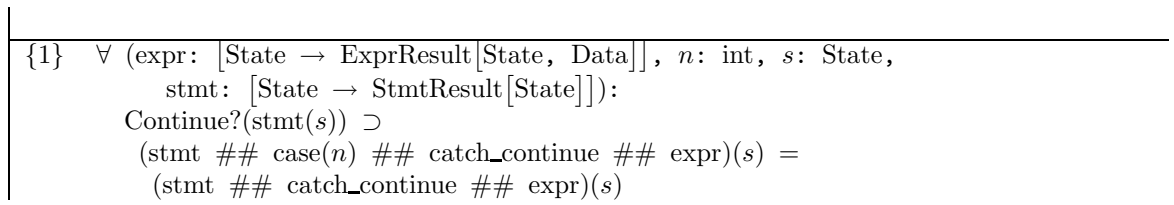
{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt, stmt1: [State → StmtResult[State]]):
Continue?(stmt(s)) ⊃
(stmt ## lift(stmt1) ## catch_continue ## expr)(s) =
(stmt ## catch_continue ## expr)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `continue_break_lift_stmt_expr`.
Q.E.D.

C.204.13 Single_Statement_Rewrites1.continue_break_case_expr

Terse proof for `continue_break_case_expr`.

`continue_break_case_expr`:



Trying repeated skolemization, instantiation, and if-lifting,

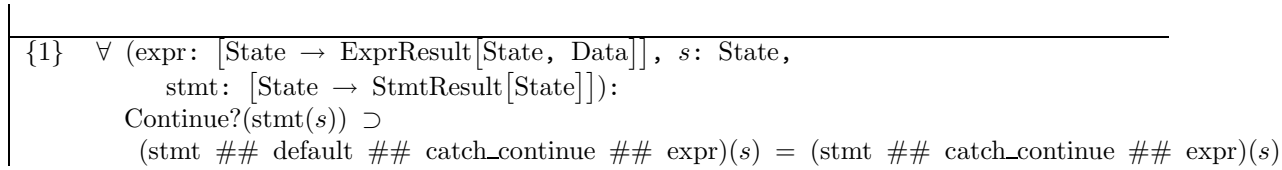
This completes the proof of `continue_break_case_expr`.

Q.E.D.

C.204.14 Single_Statement_Rewrites1.continue_break_default_expr

Terse proof for `continue_break_default_expr`.

`continue_break_default_expr`:



Trying repeated skolemization, instantiation, and if-lifting,

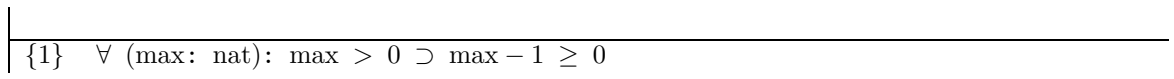
This completes the proof of `continue_break_default_expr`.

Q.E.D.

C.204.15 Single_Statement_Rewrites1.while_unroll_expr_TCC1

Terse proof for `while_unroll_expr_TCC1`.

`while_unroll_expr_TCC1`:



Repeatedly Skolemizing and flattening,

Simplifying, rewriting, and recording with decision procedures,

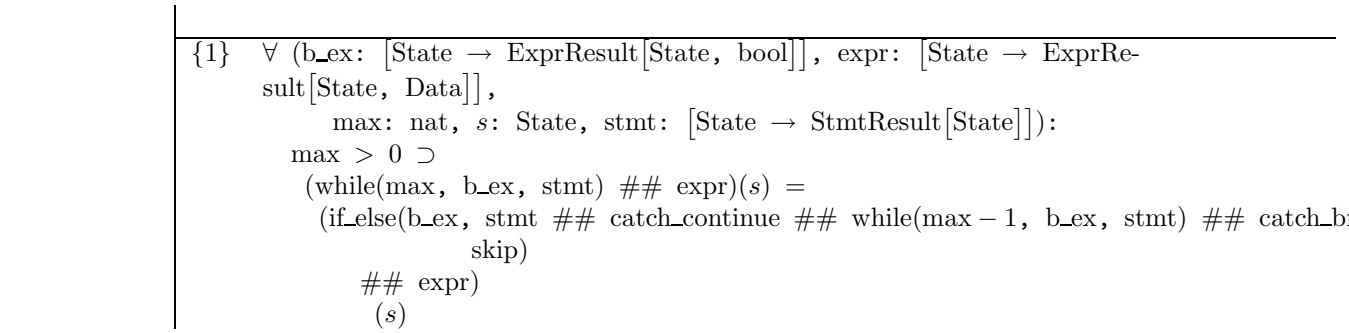
This completes the proof of `while_unroll_expr_TCC1`.

Q.E.D.

C.204.16 Single_Statement_Rewrites1.while_unroll_expr

Terse proof for `while_unroll_expr`.

while_unroll_expr:



Repeatedly Skolemizing and flattening,

Expanding the definition of ##,

Rewriting using while_unroll, matching in *,

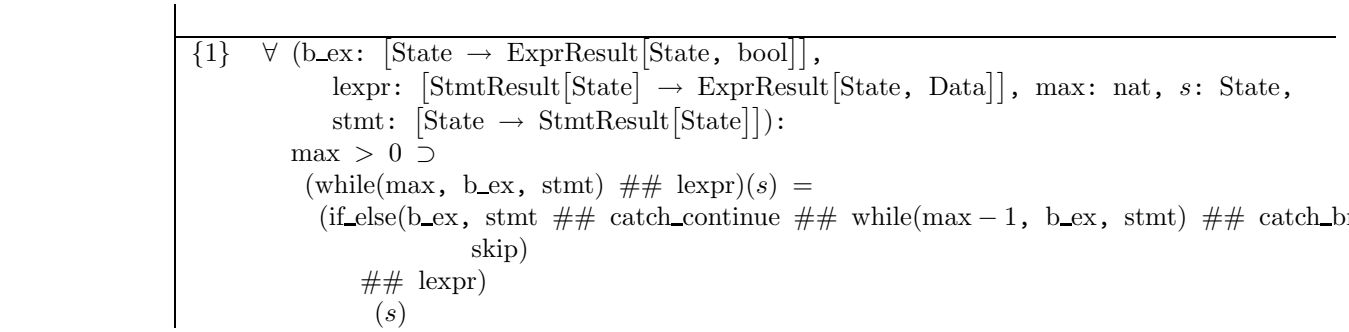
This completes the proof of while_unroll_expr.

Q.E.D.

C.204.17 Single_Statement_Rewrites1.while_unroll_lexpr

Terse proof for while_unroll_lexpr.

while_unroll_lexpr:



Repeatedly Skolemizing and flattening,

Expanding the definition of ##,

Rewriting using while_unroll, matching in *,

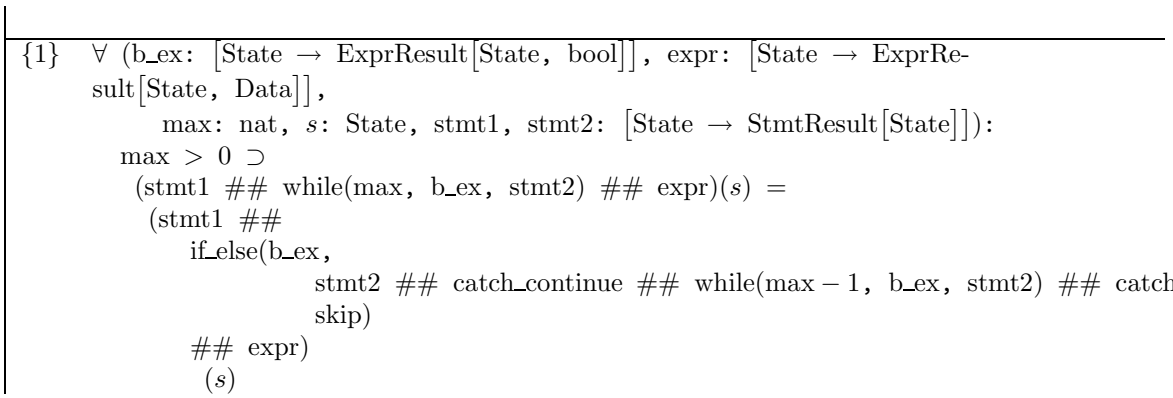
This completes the proof of while_unroll_lexpr.

Q.E.D.

C.204.18 Single_Statement_Rewrites1.stmt_while_unroll_expr

Terse proof for stmt_while_unroll_expr.

`stmt_while_unroll_expr`:

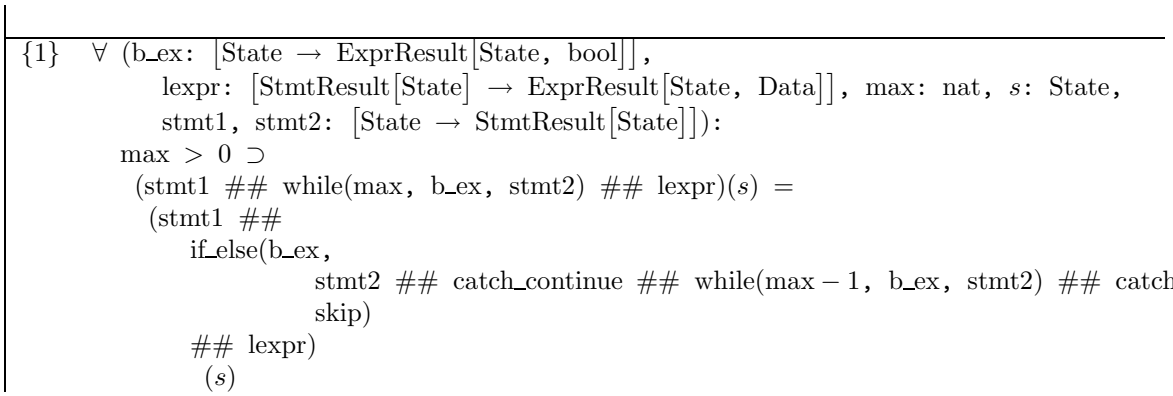


Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of while,
Using lemma while_as_if_while,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `stmt_while_unroll_expr`.
Q.E.D.

C.204.19 Single_Statement_Rewrites1.stmt_while_unroll_lexpr

Terse proof for `stmt_while_unroll_lexpr`.

`stmt_while_unroll_lexpr`:



Repeatedly Skolemizing and flattening,
Expanding the definition of ##,
Expanding the definition of while,
Using lemma while_as_if_while,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `stmt_while_unroll_lexpr`.
Q.E.D.

C.204.20 Single_Statement_Rewrites1.e2s_ok

Terse proof for `e2s_ok`.

e2s_ok:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State):
       OK?(expr(s)) ⊃ OK?(e2s(expr)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of e2s_ok.
Q.E.D.

C.204.21 Single_Statement_Rewrites1.stmt_e2s_ok

Terse proof for stmt_e2s_ok.
stmt_e2s_ok:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State,
           stmt: [State → StmtResult[State]]):
       OK?((stmt ## expr)(s)) ⊃ OK?((stmt ## e2s(expr))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of stmt_e2s_ok.
Q.E.D.

C.204.22 Single_Statement_Rewrites1.e2s_state_TCC1

Terse proof for e2s_state_TCC1.
e2s_state_TCC1:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State):
       OK?(expr(s)) ⊃
         OK?[State](e2s[State, Data](expr)(s)) ∨
           Break?[State](e2s[State, Data](expr)(s)) ∨
             Continue?[State](e2s[State, Data](expr)(s)) ∨
               Return?[State](e2s[State, Data](expr)(s)) ∨
                 Switch?[State](e2s[State, Data](expr)(s)) ∨
                   Default?[State](e2s[State, Data](expr)(s)) ∨
                     Exception?[State](e2s[State, Data](expr)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of e2s_state_TCC1.
Q.E.D.

C.204.23 Single_Statement_Rewrites1.e2s_state_TCC2

Terse proof for e2s_state_TCC2.
e2s_state_TCC2:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State):
       OK?(expr(s)) ⊃ OK?[State, Data](expr(s)) ∨ Exception?[State, Data](expr(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of e2s_state_TCC2.

Q.E.D.

C.204.24 Single_Statement_Rewrites1.e2s_state

Terse proof for `e2s_state`.

`e2s_state`:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State):
       OK?(expr(s)) ⊃ state(e2s(expr)(s)) = state(expr(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `e2s_state`.
Q.E.D.

C.204.25 Single_Statement_Rewrites1.stmt_e2s_state_TCC1

Terse proof for `stmt_e2s_state_TCC1`.

`stmt_e2s_state_TCC1`:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State,
          stmt: [State → StmtResult[State]]):
       OK?((stmt ## expr)(s)) ⊃
        OK?[State]((##[State](stmt, e2s[State, Data](expr)))(s)) ∨
         Break?[State]((##[State](stmt, e2s[State, Data](expr)))(s)) ∨
          Continue?[State]((##[State](stmt, e2s[State, Data](expr)))(s)) ∨
           Return?[State]((##[State](stmt, e2s[State, Data](expr)))(s)) ∨
            Switch?[State]((##[State](stmt, e2s[State, Data](expr)))(s)) ∨
             Default?[State]((##[State](stmt, e2s[State, Data](expr)))(s)) ∨
              Exception?[State]((##[State](stmt, e2s[State, Data](expr)))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_e2s_state_TCC1`.
Q.E.D.

C.204.26 Single_Statement_Rewrites1.stmt_e2s_state_TCC2

Terse proof for `stmt_e2s_state_TCC2`.

`stmt_e2s_state_TCC2`:

```
{1}  ∀ (expr: [State → ExprResult[State, Data]], s: State,
          stmt: [State → StmtResult[State]]):
       OK?((stmt ## expr)(s)) ⊃
        OK?[State, Data]((##[State, Data](stmt, expr))(s)) ∨
         Exception?[State, Data]((##[State, Data](stmt, expr))(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_e2s_state_TCC2`.
Q.E.D.

C.204.27 Single_Statement_Rewrites1.stmt_e2s_state

Terse proof for `stmt_e2s_state`.

`stmt_e2s_state:`

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃
state((stmt ## e2s(expr))(s)) = state((stmt ## expr)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_e2s_state`.
Q.E.D.

C.204.28 Single_Statement_Rewrites1.comp_eval_if_ok_fstmt

Terse proof for `comp_eval_if_ok_fstmt`.

`comp_eval_if_ok_fstmt:`

{1} ∀ (expr: [State → ExprResult[State, Data]],
fstmt: [Data → [State → StmtResult[State]]], s: State):
OK?(expr(s)) ⊃
(expr ## fstmt)(s) = (e2s(expr) ## fstmt(data(expr(s))))(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_eval_if_ok_fstmt`.
Q.E.D.

C.204.29 Single_Statement_Rewrites1.stmt_eval_if_ok_fstmt

Terse proof for `stmt_eval_if_ok_fstmt`.

`stmt_eval_if_ok_fstmt:`

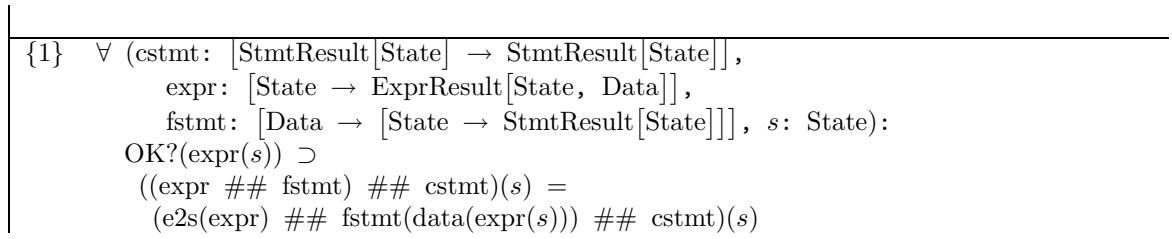
{1} ∀ (expr: [State → ExprResult[State, Data]],
fstmt: [Data → [State → StmtResult[State]]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃
(stmt ## (expr ## fstmt))(s) =
(stmt ## (e2s(expr) ## fstmt(data((stmt ## expr)(s)))))(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_eval_if_ok_fstmt`.
Q.E.D.

C.204.30 Single_Statement_Rewrites1.comp_eval_if_ok_fstmt_cstmt

Terse proof for `comp_eval_if_ok_fstmt_cstmt`.

comp_eval_if_ok_fstmt_cstmt:

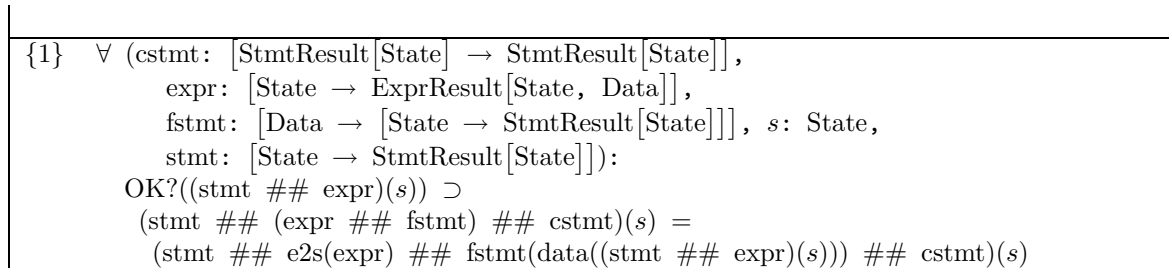


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of comp_eval_if_ok_fstmt_cstmt.
Q.E.D.

C.204.31 Single_Statement_Rewrites1.stmt_eval_if_ok_fstmt_cstmt

Terse proof for stmt_eval_if_ok_fstmt_cstmt.

stmt_eval_if_ok_fstmt_cstmt:

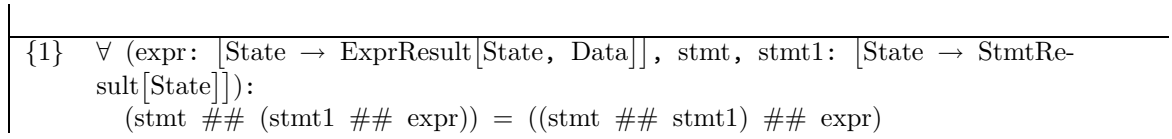


Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of stmt_eval_if_ok_fstmt_cstmt.
Q.E.D.

C.204.32 Single_Statement_Rewrites1.composition_assoc_rewrite_expr_1

Terse proof for composition_assoc_rewrite_expr_1.

composition_assoc_rewrite_expr_1:

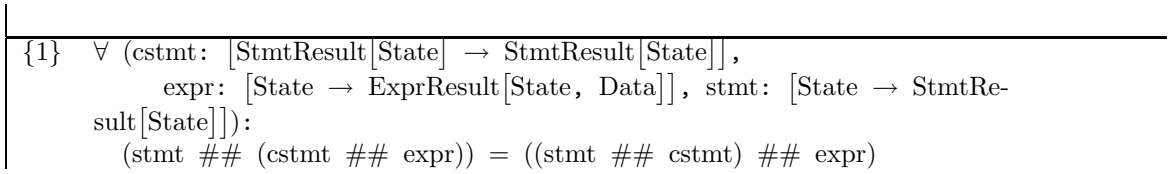


Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_expr_1.
Q.E.D.

C.204.33 Single_Statement_Rewrites1.composition_assoc_rewrite_expr_2

Terse proof for composition_assoc_rewrite_expr_2.

composition_assoc_rewrite_expr_2:

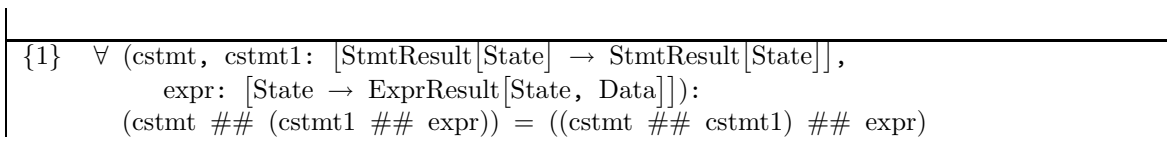


Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_expr_2.
Q.E.D.

C.204.34 Single_Statement_Rewrites1.composition_assoc_rewrite_expr_3

Terse proof for composition_assoc_rewrite_expr_3.

composition_assoc_rewrite_expr_3:

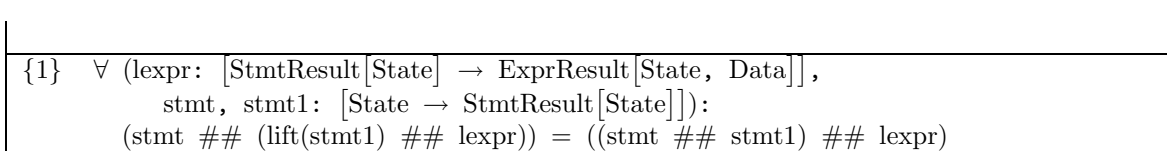


Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_expr_3.
Q.E.D.

C.204.35 Single_Statement_Rewrites1.composition_assoc_rewrite_expr_4

Terse proof for composition_assoc_rewrite_expr_4.

composition_assoc_rewrite_expr_4:

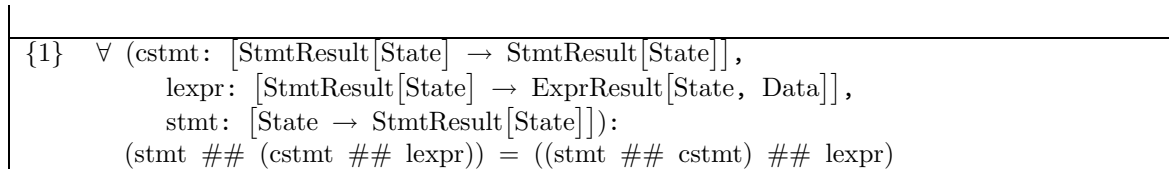


,
Applying decompose-equality,
This completes the proof of composition_assoc_rewrite_expr_4.
Q.E.D.

C.204.36 Single_Statement_Rewrites1.composition_assoc_rewrite_expr_5

Terse proof for composition_assoc_rewrite_expr_5.

composition_assoc_rewrite_expr_5:

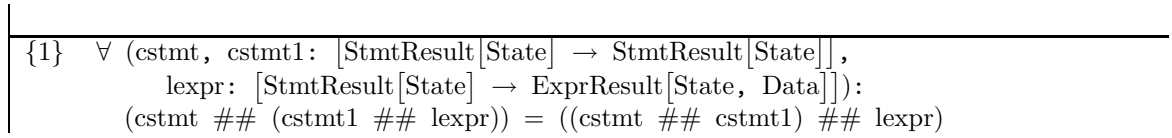


Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_expr_5.
Q.E.D.

C.204.37 Single_Statement_Rewrites1.composition_assoc_rewrite_expr_6

Terse proof for composition_assoc_rewrite_expr_6.

composition_assoc_rewrite_expr_6:

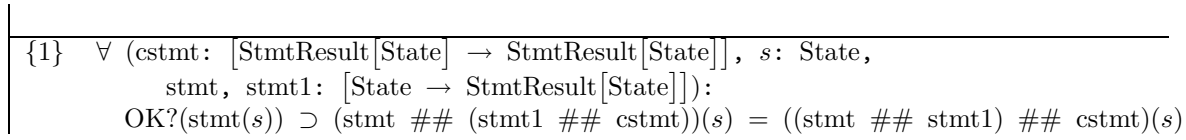


Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_expr_6.
Q.E.D.

C.204.38 Single_Statement_Rewrites1.composition_assoc_rewrite_stmt_ok

Terse proof for composition_assoc_rewrite_stmt_ok.

composition_assoc_rewrite_stmt_ok:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of composition_assoc_rewrite_stmt_ok.
Q.E.D.

C.204.39 Single_Statement_Rewrites1.composition_assoc_rewrite_stmt_ok_expr

Terse proof for composition_assoc_rewrite_stmt_ok_expr.

composition_assoc_rewrite_stmt_ok_expr:

```
{1}  ∀ (cstmt: [StmtResult[State] → StmtResult[State]],
          expr: [State → ExprResult[State, Data]], s: State,
          stmt, stmt1: [State → StmtResult[State]]):
        OK?(stmt(s)) ⊃
         (stmt ## (stmt1 ## cstmt) ## expr)(s) = ((stmt ## stmt1) ## cstmt ## expr)(s)
```

,
This completes the proof of composition_assoc_rewrite_stmt_ok_expr.
Q.E.D.

C.204.40 Single_Statement_Rewrites1.composition_assoc_rewrite_stmt_ok_expr_lexpr

Terse proof for composition_assoc_rewrite_stmt_ok_expr_lexpr.
composition_assoc_rewrite_stmt_ok_expr_lexpr:

```
{1}  ∀ (cstmt: [StmtResult[State] → StmtResult[State]],
          lexpr: [StmtResult[State] → ExprResult[State, Data]], s: State,
          stmt, stmt1: [State → StmtResult[State]]):
        OK?(stmt(s)) ⊃
         (stmt ## (stmt1 ## cstmt) ## lexpr)(s) = ((stmt ## stmt1) ## cstmt ## lexpr)(s)
```

,
This completes the proof of composition_assoc_rewrite_stmt_ok_expr_lexpr.
Q.E.D.

C.204.41 Single_Statement_Rewrites1.ok_result_fstmt

Terse proof for ok_result_fstmt.
ok_result_fstmt:

```
{1}  ∀ (fstmt: [Data → [State → StmtResult[State]]], data: Data):
        ok_result(data) ## fstmt = fstmt(data)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of ok_result_fstmt.
Q.E.D.

C.204.42 Single_Statement_Rewrites1.pm_q_prop_read_ok_s_read_ok

Terse proof for pm_q_prop_read_ok_s_read_ok.
pm_q_prop_read_ok_s_read_ok:

```
{1}  ∀ (addr: Address, dt: (interpreted_data_type?[Data]), pm: Plain_Memory[State], s: State):
        pm_q_prop_read(pm, dt, addr)(skip(s)) ⊃ OK?(read_data(pm, dt)(addr)(s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of pm_q_prop_read_ok_s_read_ok.

Q.E.D.

C.205 Proofs for Single_Statement_Rewrites2 (statement-rewrites.pvs)

C.205.1 Single_Statement_Rewrites2.ok_result_fexpr

Terse proof for `ok_result_fexpr`.
`ok_result_fexpr:`

{1} ∀ (fexpr: [Data2 → [State → ExprResult[State, Data1]]], data: Data2):
ok_result(data) ## fexpr = fexpr(data)

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_fexpr`.
Q.E.D.

C.205.2 Single_Statement_Rewrites2.e2s_merge

Terse proof for `e2s_merge`.
`e2s_merge:`

{1} ∀ (expr: [State → ExprResult[State, Data1]],
expr1: [State → ExprResult[State, Data2]]):
(e2s(expr) ## e2s(expr1)) = e2s(expr ## expr1)

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `e2s_merge`.
Q.E.D.

C.205.3 Single_Statement_Rewrites2.stmt_e2s_merge

Terse proof for `stmt_e2s_merge`.
`stmt_e2s_merge:`

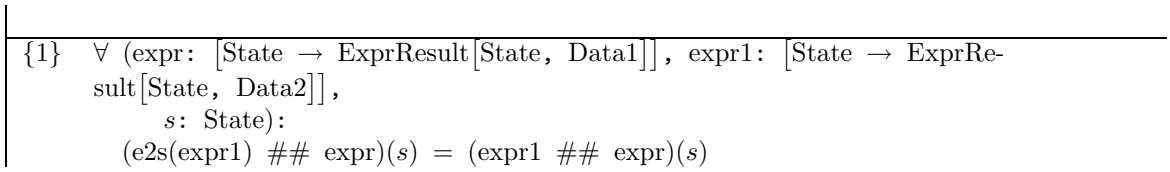
{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
stmt: [State → StmtResult[State]]):
(stmt ## e2s(expr) ## e2s(expr1)) = (stmt ## e2s(expr ## expr1))

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_e2s_merge`.
Q.E.D.

C.205.4 Single_Statement_Rewrites2.e2s_expr

Terse proof for `e2s_expr`.

`e2s_expr:`



$\{1\}$ $\forall$ (expr: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data1}]]$, expr1: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data2}]]$,
s: State):
$(\text{e2s(expr1)} \mathbin{\#\#} \text{expr})(s) = (\text{expr1} \mathbin{\#\#} \text{expr})(s)$

,

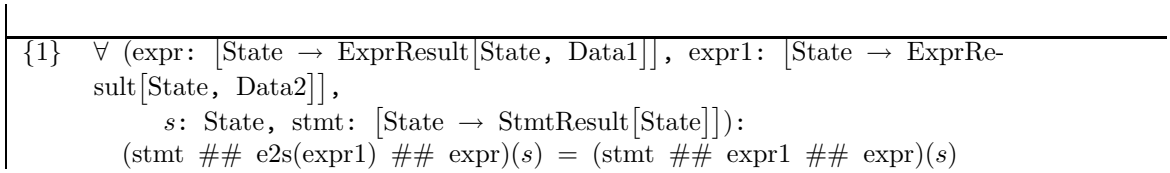
This completes the proof of `e2s_expr`.

Q.E.D.

C.205.5 Single_Statement_Rewrites2.stmt_e2s_expr

Terse proof for `stmt_e2s_expr`.

`stmt_e2s_expr:`



$\{1\}$ $\forall$ (expr: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data1}]]$, expr1: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data2}]]$,
s: State, stmt: $[\text{State} \to \text{StmtResult}[\text{State}]]$):
$(\text{stmt} \mathbin{\#\#} \text{e2s(expr1)} \mathbin{\#\#} \text{expr})(s) = (\text{stmt} \mathbin{\#\#} \text{expr1} \mathbin{\#\#} \text{expr})(s)$

,

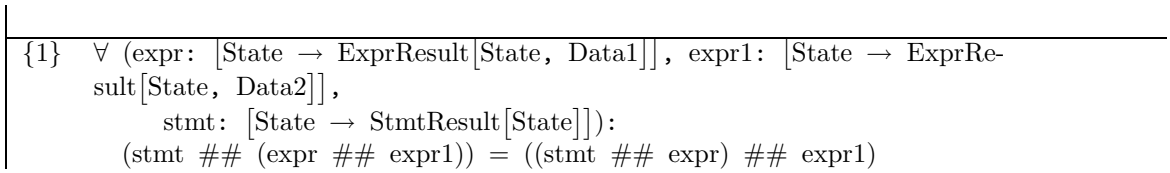
This completes the proof of `stmt_e2s_expr`.

Q.E.D.

C.205.6 Single_Statement_Rewrites2.composition_assoc_expression_rewrite_stmt

Terse proof for `composition_assoc_expression_rewrite_stmt`.

`composition_assoc_expression_rewrite_stmt:`



$\{1\}$ $\forall$ (expr: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data1}]]$, expr1: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data2}]]$,
stmt: $[\text{State} \to \text{StmtResult}[\text{State}]]$):
$(\text{stmt} \mathbin{\#\#} (\text{expr} \mathbin{\#\#} \text{expr1})) = ((\text{stmt} \mathbin{\#\#} \text{expr}) \mathbin{\#\#} \text{expr1})$

Repeatedly Skolemizing and flattening,

Applying decompose-equality,

Trying repeated skolemization, instantiation, and if-lifting,

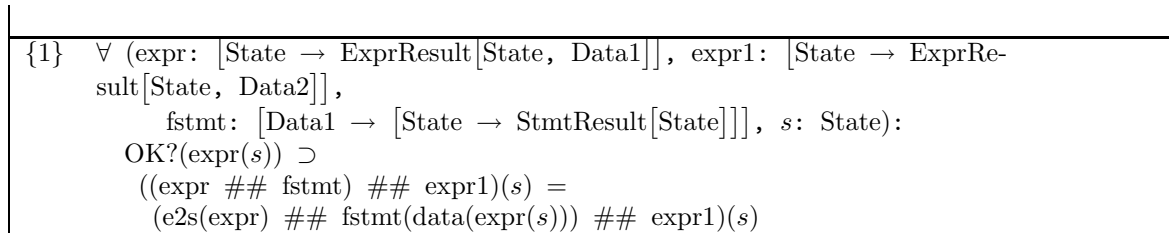
This completes the proof of `composition_assoc_expression_rewrite_stmt`.

Q.E.D.

C.205.7 Single_Statement_Rewrites2.comp_eval_if_ok_fstmt_expr

Terse proof for `comp_eval_if_ok_fstmt_expr`.

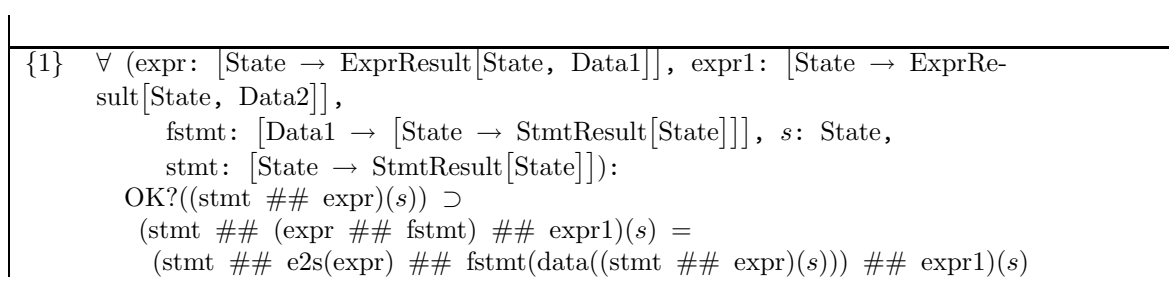
comp_eval_if_ok_fstmt_expr:



,
This completes the proof of comp_eval_if_ok_fstmt_expr.
Q.E.D.

C.205.8 Single_Statement_Rewrites2.stmt_eval_if_ok_fstmt_expr

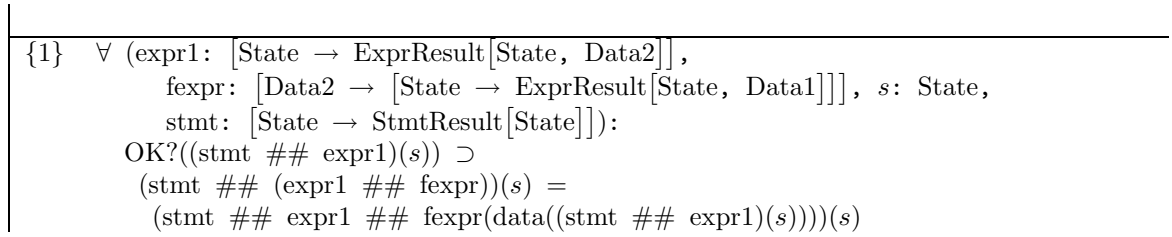
Terse proof for stmt_eval_if_ok_fstmt_expr.
stmt_eval_if_ok_fstmt_expr:



,
This completes the proof of stmt_eval_if_ok_fstmt_expr.
Q.E.D.

C.205.9 Single_Statement_Rewrites2.stmt_eval_if_ok_fexpr

Terse proof for stmt_eval_if_ok_fexpr.
stmt_eval_if_ok_fexpr:



,
This completes the proof of stmt_eval_if_ok_fexpr.
Q.E.D.

C.205.10 Single_Statement_Rewrites2.comp_eval_if_ok_fstmt_cstmt_expr

Terse proof for comp_eval_if_ok_fstmt_cstmt_expr.

comp_eval_if_ok_fstmt_cstmt_expr:

{1} ∀ (cstmt: [StmtResult[State] → StmtResult[State]],
expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
fstmt: [Data1 → [State → StmtResult[State]]], s: State):
OK?(expr(s)) ⊃
(((expr ## fstmt) ## cstmt) ## expr1)(s) =
(((e2s(expr) ## fstmt(data(expr(s)))) ## cstmt) ## expr1)(s)

,
This completes the proof of comp_eval_if_ok_fstmt_cstmt_expr.
Q.E.D.

C.205.11 Single_Statement_Rewrites2.stmt_eval_if_ok_fstmt_cstmt_expr

Terse proof for stmt_eval_if_ok_fstmt_cstmt_expr.
stmt_eval_if_ok_fstmt_cstmt_expr:

{1} ∀ (cstmt: [StmtResult[State] → StmtResult[State]],
expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
fstmt: [Data1 → [State → StmtResult[State]]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃
(stmt ## (expr ## fstmt) ## cstmt ## expr1)(s) =
(stmt ## e2s(expr) ## fstmt(data((stmt ## expr)(s))) ## cstmt ## expr1)(s)

,
This completes the proof of stmt_eval_if_ok_fstmt_cstmt_expr.
Q.E.D.

C.206 Proofs for Single_Statement_Rewrites3 (statement-rewrites.pvs)

C.206.1 Single_Statement_Rewrites3.ok_result_elemination_2

Terse proof for ok_result_elemination_2.
ok_result_elemination_2:

{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
data: Data3):
((expr ## ok_result[State, Data3](data)) ## expr1) = (expr ## expr1)

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of ok_result_elemination_2.

Q.E.D.

C.206.2 Single_Statement_Rewrites3.comp_eval_if_ok_fexpr_expr

Terse proof for `comp_eval_if_ok_fexpr_expr`.

`comp_eval_if_ok_fexpr_expr`:

{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
fexpr: [Data1 → [State → ExprResult[State, Data3]]], s: State):
OK?(expr(s)) ⊃
((expr ## fexpr) ## expr1)(s) = (expr ## fexpr(data(expr(s))) ## expr1)(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `comp_eval_if_ok_fexpr_expr`.
Q.E.D.

C.206.3 Single_Statement_Rewrites3.expr_eval_if_ok_fexpr

Terse proof for `expr_eval_if_ok_fexpr`.

`expr_eval_if_ok_fexpr`:

{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
fexpr: [Data1 → [State → ExprResult[State, Data3]]], s: State):
OK?((expr1 ## expr)(s)) ⊃
(expr1 ## (expr ## fexpr))(s) =
(expr1 ## expr ## fexpr(data((expr1 ## expr)(s))))(s)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_eval_if_ok_fexpr`.
Q.E.D.

C.206.4 Single_Statement_Rewrites3.composition_assoc_expression_rewrite

Terse proof for `composition_assoc_expression_rewrite`.

`composition_assoc_expression_rewrite`:

{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
expr2: [State → ExprResult[State, Data3]]):
(expr ## (expr1 ## expr2)) = ((expr ## expr1) ## expr2)

Trying repeated skolemization, instantiation, and if-lifting,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_expression_rewrite`.
Q.E.D.

C.206.5 Single_Statement_Rewrites3.stmt_eval_if_ok_fexpr_expr

Terse proof for `stmt_eval_if_ok_fexpr_expr`.

`stmt_eval_if_ok_fexpr_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
fexpr: [Data1 → [State → ExprResult[State, Data3]]], s: State,
stmt: [State → StmtResult[State]]):
OK?((stmt ## expr)(s)) ⊃
(stmt ## (expr ## fexpr) ## expr1)(s) =
(stmt ## expr ## fexpr(data((stmt ## expr)(s))) ## expr1)(s)

,

This completes the proof of `stmt_eval_if_ok_fexpr_expr`.

Q.E.D.

C.207 Proofs for Single_Statement_Rewrites4 (statement-rewrites.pvs)

C.207.1 Single_Statement_Rewrites4.expr_eval_if_ok_fexpr_expr

Terse proof for `expr_eval_if_ok_fexpr_expr`.

`expr_eval_if_ok_fexpr_expr:`

{1} ∀ (expr: [State → ExprResult[State, Data1]], expr1: [State → ExprResult[State, Data2]],
expr2: [State → ExprResult[State, Data4]],
fexpr: [Data2 → [State → ExprResult[State, Data3]]], s: State):
OK?((expr ## expr1)(s)) ⊃
(expr ## (expr1 ## fexpr) ## expr2)(s) =
(expr ## expr1 ## fexpr(data((expr ## expr1)(s))) ## expr2)(s)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `expr_eval_if_ok_fexpr_expr`.

Q.E.D.

C.208 Proofs for SkipStatements (statements.pvs)

This theory contains no provable formal statements.

C.209 Proofs for State_Transformer (state-transformer.pvs)

This theory contains no provable formal statements.

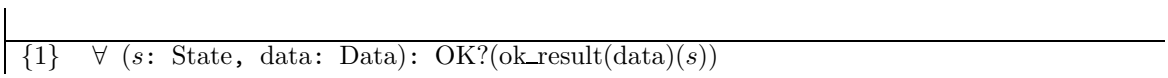
C.210 Proofs for State_Transformer_Lift (state-transformer.pvs)

This theory contains no provable formal statements.

C.211 Proofs for State_Transformer_Lift_Expr (state-transformer.pvs)

C.211.1 State_Transformer_Lift_Expr.ok_result_ok

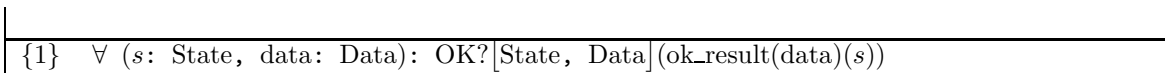
Terse proof for `ok_result_ok`.
`ok_result_ok`:



,
This completes the proof of `ok_result_ok`.
Q.E.D.

C.211.2 State_Transformer_Lift_Expr.ok_result_data_TCC1

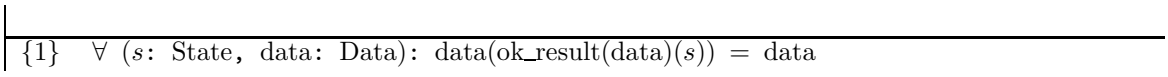
Terse proof for `ok_result_data_TCC1`.
`ok_result_data_TCC1`:



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `ok_result_data_TCC1`.
Q.E.D.

C.211.3 State_Transformer_Lift_Expr.ok_result_data

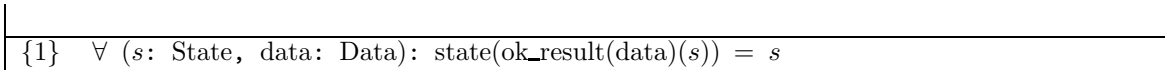
Terse proof for `ok_result_data`.
`ok_result_data`:



,
This completes the proof of `ok_result_data`.
Q.E.D.

C.211.4 State_Transformer_Lift_Expr.ok_result_state

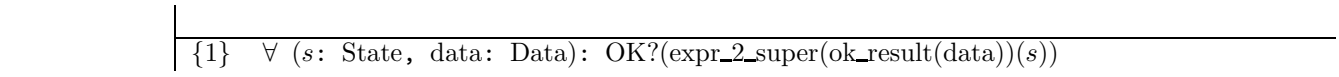
Terse proof for `ok_result_state`.
`ok_result_state`:



,
This completes the proof of `ok_result_state`.
Q.E.D.

C.211.5 State_Transformer_Lift_Expr.expr_2_super_ok_result

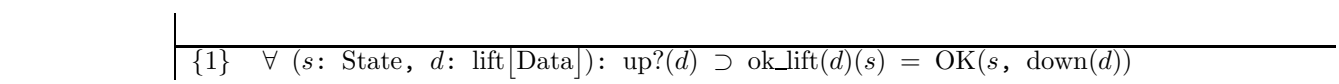
Terse proof for `expr_2_super_ok_result`.
`expr_2_super_ok_result:`



,
This completes the proof of `expr_2_super_ok_result`.
Q.E.D.

C.211.6 State_Transformer_Lift_Expr.ok_lift_ok

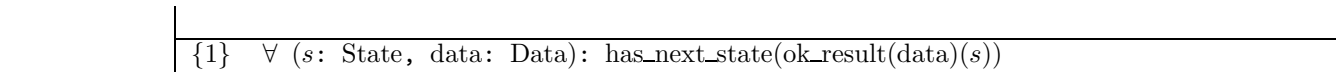
Terse proof for `ok_lift_ok`.
`ok_lift_ok:`



,
This completes the proof of `ok_lift_ok`.
Q.E.D.

C.211.7 State_Transformer_Lift_Expr.has_next_state_ok_result

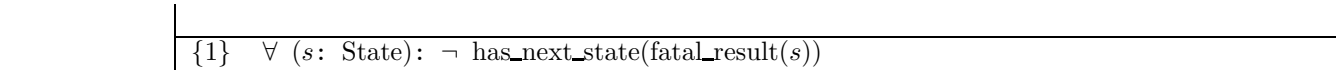
Terse proof for `has_next_state_ok_result`.
`has_next_state_ok_result:`



,
This completes the proof of `has_next_state_ok_result`.
Q.E.D.

C.211.8 State_Transformer_Lift_Expr.has_next_state_fatal_result

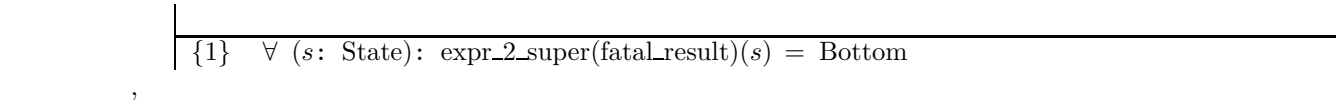
Terse proof for `has_next_state_fatal_result`.
`has_next_state_fatal_result:`



,
This completes the proof of `has_next_state_fatal_result`.
Q.E.D.

C.211.9 State_Transformer_Lift_Expr.expr_2_super_fatal_result

Terse proof for `expr_2_super_fatal_result`.
`expr_2_super_fatal_result:`



,
This completes the proof of `expr_2_super_fatal_result`.
Q.E.D.

C.212 Proofs for State_Transformer_Lift_Stmt (state-transformer.pvs)

This theory contains no provable formal statements.

C.213 Proofs for Statement_Composition_Rewrites (state-transformer.pvs)

C.213.1 Statement_Composition_Rewrites.composition_assoc_rewrite_1

Terse proof for `composition_assoc_rewrite_1`.

`composition_assoc_rewrite_1:`

```
{1}  ∀ (stmt_1, stmt_2, stmt_3: [State → StmtResult[State]]):
       (stmt_1 ## (stmt_2 ## stmt_3)) = ((stmt_1 ## stmt_2) ## stmt_3)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_1`.
Q.E.D.

C.213.2 Statement_Composition_Rewrites.composition_assoc_rewrite_2

Terse proof for `composition_assoc_rewrite_2`.

`composition_assoc_rewrite_2:`

```
{1}  ∀ (lift_stmt_3: [StmtResult[State] → StmtResult[State]],
          stmt_1, stmt_2: [State → StmtResult[State]]):
       (stmt_1 ## (lift(stmt_2) ## lift_stmt_3)) = ((stmt_1 ## stmt_2) ## lift_stmt_3)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_2`.
Q.E.D.

C.213.3 Statement_Composition_Rewrites.composition_assoc_rewrite_3

Terse proof for `composition_assoc_rewrite_3`.

`composition_assoc_rewrite_3:`

```
{1}  ∀ (lift_stmt_2: [StmtResult[State] → StmtResult[State]],
          stmt_1, stmt_3: [State → StmtResult[State]]):
       (stmt_1 ## (lift_stmt_2 ## stmt_3)) = ((stmt_1 ## lift_stmt_2) ## stmt_3)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_3`.
Q.E.D.

C.213.4 Statement_Composition_Rewrites.composition_assoc_rewrite_4

Terse proof for `composition_assoc_rewrite_4`.
`composition_assoc_rewrite_4:`

```
{1}  ∀ (lift_stmt_2, lift_stmt_3: [StmtResult[State] → StmtResult[State]],
          stmt_1: [State → StmtResult[State]]):
        (stmt_1 ## (lift_stmt_2 ## lift_stmt_3)) = ((stmt_1 ## lift_stmt_2) ## lift_stmt_3)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_4`.
Q.E.D.

C.213.5 Statement_Composition_Rewrites.composition_assoc_rewrite_5

Terse proof for `composition_assoc_rewrite_5`.
`composition_assoc_rewrite_5:`

```
{1}  ∀ (lift_stmt_1: [StmtResult[State] → StmtResult[State]],
          stmt_2, stmt_3: [State → StmtResult[State]]):
        (lift_stmt_1 ## (stmt_2 ## stmt_3)) = ((lift_stmt_1 ## stmt_2) ## stmt_3)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_5`.
Q.E.D.

C.213.6 Statement_Composition_Rewrites.composition_assoc_rewrite_6

Terse proof for `composition_assoc_rewrite_6`.
`composition_assoc_rewrite_6:`

```
{1}  ∀ (lift_stmt_1, lift_stmt_3: [StmtResult[State] → StmtResult[State]],
          stmt_2: [State → StmtResult[State]]):
        (equalities[[StmtResult[State] → StmtResult[State]]].=
              ((lift_stmt_1 ## (lift(stmt_2) ## lift_stmt_3)),
                ((lift_stmt_1 ## stmt_2) ## lift_stmt_3)))
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_6`.
Q.E.D.

C.213.7 Statement_Composition_Rewrites.composition_assoc_rewrite_7

Terse proof for `composition_assoc_rewrite_7`.
`composition_assoc_rewrite_7:`

```
{1}  ∀ (lift_stmt_1, lift_stmt_2: [StmtResult[State] → StmtResult[State]],
            stmt_3: [State → StmtResult[State]]):
       (lift_stmt_1 ## (lift_stmt_2 ## stmt_3)) = ((lift_stmt_1 ## lift_stmt_2) ## stmt_3)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_7`.
Q.E.D.

C.213.8 Statement_Composition_Rewrites.composition_assoc_rewrite_8

Terse proof for `composition_assoc_rewrite_8`.
`composition_assoc_rewrite_8:`

```
{1}  ∀ (lift_stmt_1, lift_stmt_2, lift_stmt_3: [StmtResult[State] → StmtResult[State]]):
       (equalities[[StmtResult[State] → StmtResult[State]]] .=
             ((lift_stmt_1 ## (lift_stmt_2 ## lift_stmt_3)),
               ((lift_stmt_1 ## lift_stmt_2) ## lift_stmt_3)))
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `composition_assoc_rewrite_8`.
Q.E.D.

C.214 Proofs for Statement_Rewrites (statement-rewrites.pvs)

This theory contains no provable formal statements.

C.215 Proofs for Statements (statements.pvs)

This theory contains no provable formal statements.

C.216 Proofs for StmtResult (result.pvs)

This theory contains no provable formal statements.

C.217 Proofs for StmtResult_adt (StmtResult_adt.pvs)

This theory contains no provable formal statements.

C.218 Proofs for StmtResult_adt_map (StmtResult_adt.pvs)

This theory contains no provable formal statements.

C.219 Proofs for StmtResult_adt_reduce (StmtResult_adt.pvs)

This theory contains no provable formal statements.

C.220 Proofs for Subtype_stable (graph.pvs)

C.220.1 Subtype_stable.length_stable

Terse proof for `length_stable`.

`length_stable:`

$$\{1\}\quad \forall\ (P\colon\ \mathrm{PRED}[T],\ l\colon\ \mathrm{list}[T])\colon \\ \quad \mathrm{every}(P)(l)\ \supset\ \mathrm{length}[(P)](l)\ =\ \mathrm{length}[T](l)$$

By induction on l, and by repeatedly rewriting and simplifying,
This completes the proof of `length_stable`.
Q.E.D.

C.220.2 Subtype_stable.nth_stable_TCC1

Terse proof for `nth_stable_TCC1`.

`nth_stable_TCC1:`

$$\{1\}\quad \forall\ (P\colon\ \mathrm{PRED}[T],\ l\colon\ \mathrm{list}[T],\ n\colon\ \mathrm{below}(\mathrm{length}[T](l)))\colon \\ \quad \mathrm{every}(P)(l)\ \supset\ n\ <\ \mathrm{length}[(P)](l)$$

Repeatedly Skolemizing and flattening,
Rewriting using length_stable, matching in *,
This completes the proof of `nth_stable_TCC1`.
Q.E.D.

C.220.3 Subtype_stable.nth_stable

Terse proof for `nth_stable`.

`nth_stable:`

{1} $\forall$ (P: PRED$[T]$, l: list$[T]$, n: below(length(l))):
every(P)(l) $\supset$ nth$[(P)](l, n)$ = nth$[T](l, n)$

Installing automatic rewrites from: length_stable
Inducting on l on formula 1,
we get 3 subgoals:

`nth_stable.1:`

{1} $\forall$ (P: PRED$[T]$, n: below(length(null))):
every(P)(null) $\supset$ nth$[(P)]$(null, n) = nth$[T]$(null, n)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `nth_stable.1`.

`nth_stable.2:`

{1} $\forall$ (cons1_var: T, cons2_var: list$[T]$):
($\forall$ (P: PRED$[T]$, n: below(length(cons2_var))):
every(P)(cons2_var) $\supset$ nth$[(P)]$(cons2_var, n) = nth$[T]$(cons2_var, n))
$\supset$
($\forall$ (P: PRED$[T]$, n: below(length(cons(cons1_var, cons2_var)))):
every(P)(cons(cons1_var, cons2_var)) $\supset$
nth$[(P)]$(cons(cons1_var, cons2_var), n) =
nth$[T]$(cons(cons1_var, cons2_var), n))

Repeatedly Skolemizing and flattening,
Expanding the definition of nth,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of every,
Applying disjunctive simplification to flatten sequent,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_stable.2`.

`nth_stable.3:`

{1} $\forall$ (l: list$[T]$, P: PRED$[T]$, n: below(length$[T](l)$)):
every(P)(l) $\supset$ n < length$[(P)](l)$
{2} $\forall$ (P: PRED$[T]$, n: below(length(l'))):
every(P)(l') $\supset$ nth$[(P)](l', n)$ = nth$[T](l', n)$

Hiding formulas: 2,
Repeatedly Skolemizing and flattening,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `nth_stable.3`.
Q.E.D.

C.221 Proofs for SuperResult (result.pvs)

This theory contains no provable formal statements.

C.222 Proofs for SuperResult_adt (SuperResult_adt.pvs)

This theory contains no provable formal statements.

C.223 Proofs for SuperResult_adt_map (SuperResult_adt.pvs)

This theory contains no provable formal statements.

C.224 Proofs for SuperResult_adt_reduce (SuperResult_adt.pvs)

This theory contains no provable formal statements.

C.225 Proofs for Super_Embedding (result.pvs)

This theory contains no provable formal statements.

C.226 Proofs for Super_Embedding_Expr (result.pvs)

C.226.1 Super_Embedding_Expr.expr_2_super_res_TCC1

Terse proof for `expr_2_super_res_TCC1`.
`expr_2_super_res_TCC1:`

```
{1}  ∀ (expr: ExprResult[State, Data]):
       OK?(expr) ⊃ OK?[State, Data](expr) ∨ Exception?[State, Data](expr)
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_2_super_res_TCC1`.
Q.E.D.

C.226.2 Super_Embedding_Expr.expr_2_super_res_TCC2

Terse proof for `expr_2_super_res_TCC2`.
`expr_2_super_res_TCC2:`

```
{1}  ∀ (expr: ExprResult[State, Data]):
       has_next_state(expr) ∧ ¬ OK?(expr) ⊃
        OK?[State, Data](expr) ∨ Exception?[State, Data](expr)
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `expr_2_super_res_TCC2`.
Q.E.D.

C.227 Proofs for Super_Embedding_Stmt (result.pvs)

C.227.1 Super_Embedding_Stmt.stmt_2_super_res_TCC1

Terse proof for `stmt_2_super_res_TCC1`.

`stmt_2_super_res_TCC1:`

{1} ∀ (stmt: StmtResult[State]):
OK?(stmt) ⊃
OK?[State](stmt) ∨
Break?[State](stmt) ∨
Continue?[State](stmt) ∨
Return?[State](stmt) ∨
Switch?[State](stmt) ∨ Default?[State](stmt) ∨ Exception?[State](stmt)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_2_super_res_TCC1`.
Q.E.D.

C.227.2 Super_Embedding_Stmt.stmt_2_super_res_TCC2

Terse proof for `stmt_2_super_res_TCC2`.

`stmt_2_super_res_TCC2:`

{1} ∀ (stmt: StmtResult[State]):
has_next_state(stmt) ∧ ¬ OK?(stmt) ⊃
OK?[State](stmt) ∨
Break?[State](stmt) ∨
Continue?[State](stmt) ∨
Return?[State](stmt) ∨
Switch?[State](stmt) ∨ Default?[State](stmt) ∨ Exception?[State](stmt)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `stmt_2_super_res_TCC2`.
Q.E.D.

C.228 Proofs for Super_Result_Util (result.pvs)

This theory contains no provable formal statements.

C.229 Proofs for Transformer_Invariant (state-transformer.pvs)

C.229.1 Transformer_Invariant.result_pred_TCC1

Terse proof for `result_pred_TCC1`.

`result_pred_TCC1`:

{1} ∀ (res: SuperResult[State]):
has_next_state(res) ⊃ OK?[State](res) ∨ abnormal?[State](res)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `result_pred_TCC1`.
Q.E.D.

C.229.2 Transformer_Invariant.super_transformer_invariant_next_ok_TCC1

Terse proof for `super_transformer_invariant_next_ok_TCC1`.

`super_transformer_invariant_next_ok_TCC1`:

{1} ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → SuperResult[State]]):
transformer_invariant?(states, transformers) ∧
states(s) ∧ transformers(q) ∧ has_next_state(q(s))
⊃ OK?[State](q(s)) ∨ abnormal?[State](q(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_transformer_invariant_next_ok_TCC1`.
Q.E.D.

C.229.3 Transformer_Invariant.super_transformer_invariant_next_ok

Terse proof for `super_transformer_invariant_next_ok`.

`super_transformer_invariant_next_ok`:

{1} ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → SuperResult[State]]):
transformer_invariant?(states, transformers) ∧
states(s) ∧ transformers(q) ∧ has_next_state(q(s))
⊃ states(state(q(s)))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `super_transformer_invariant_next_ok`.
Q.E.D.

C.229.4 Transformer_Invariant.transformer_invariant_mono_transformers

Terse proof for `transformer_invariant_mono_transformers`.

`transformer_invariant_mono_transformers`:

{1} ∀ (states: PRED[State],
transformers_1, transformers_2: PRED[[State → SuperResult[State]]]):
(transformers_1 ⊆ transformers_2) ∧ transformer_invariant?(states, transformers_2) ⊃
transformer_invariant?(states, transformers_1)

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `transformer_invariant_mono_transformers`.

Q.E.D.

C.229.5 Transformer_Invariant.transformer_invariant_union_transformers

Terse proof for `transformer_invariant_union_transformers`.

`transformer_invariant_union_transformers`:

{1} ∀ (states: PRED[State],
transformers_1, transformers_2: PRED[[State → SuperResult[State]]]):
transformer_invariant?(states, transformers_1) ∧
transformer_invariant?(states, transformers_2)
⊃ transformer_invariant?(states, (transformers_1 ∪ transformers_2))

Installing automatic rewrites from: member

Repeatedly Skolemizing and flattening,

Expanding the definition of transformer_invariant?,

Repeatedly Skolemizing and flattening,

Expanding the definition of union,

Splitting conjunctions,

we get 2 subgoals:

`transformer_invariant_union_transformers.1`:

{-1} (q' ∈ transformers′)
{-2} ∀ (s: State, q: [State → SuperResult[State]]):
states′(s) ∧ transformers′(q) ⊃ result_pred(states′)(q(s))
{-3} ∀ (s: State, q: [State → SuperResult[State]]):
states′(s) ∧ transformers″(q) ⊃ result_pred(states′)(q(s))
{-4} states′(s')

{1} result_pred(states′)(q'(s'))

Instantiating quantified variables,

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `transformer_invariant_union_transformers.1`.

`transformer_invariant_union_transformers.2:`

{-1}	$(q' \in \text{transformers}'')$
{-2}	$\forall$ (s: State, q: [State → SuperResult[State]]): states′(s) ∧ transformers′(q) ⊃ result_pred(states′)(q(s))
{-3}	$\forall$ (s: State, q: [State → SuperResult[State]]): states′(s) ∧ transformers″(q) ⊃ result_pred(states′)(q(s))
{-4}	states′(s')
{1}	result_pred(states′)(q'(s'))

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformer_invariant_union_transformers.2`.
Q.E.D.

C.229.6 Transformer_Invariant.transformer_invariant_truth

Terse proof for `transformer_invariant_truth`.

`transformer_invariant_truth:`

{1}	$\forall$ (transformers: PRED[[State → SuperResult[State]]]): transformer_invariant?(fullset[State], transformers)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `transformer_invariant_truth`.
Q.E.D.

C.229.7 Transformer_Invariant.transformer_invariant_all_transformers

Terse proof for `transformer_invariant_all_transformers`.

`transformer_invariant_all_transformers:`

{1}	$\forall$ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]]): ($\forall$ (q: [State → SuperResult[State]]): transformers(q) ⊃ transformer_invariant?(states, singleton(q))) ⊃ transformer_invariant?(states, transformers)

Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
Instantiating quantified variables,
Expanding the definition of singleton,
which is trivially true.
This completes the proof of `transformer_invariant_all_transformers`.
Q.E.D.

C.229.8 Transformer_Invariant.super_transformers_ok_ok

Terse proof for `super_transformers_ok_ok`.

super_transformers_ok_ok:

{1} ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → SuperResult[State]]):
transformers_ok?(states, transformers) ∧ states(s) ∧ transformers(q) ⊃ OK?(q(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of super_transformers_ok_ok.
Q.E.D.

C.229.9 Transformer_Invariant.transformers_ok_mono_transformers

Terse proof for transformers_ok_mono_transformers.
transformers_ok_mono_transformers:

{1} ∀ (states: PRED[State],
transformers_1, transformers_2: PRED[[State → SuperResult[State]]]):
(transformers_1 ⊆ transformers_2) ∧ transformers_ok?(states, transformers_2) ⊃
transformers_ok?(states, transformers_1)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of transformers_ok_mono_transformers.
Q.E.D.

C.229.10 Transformer_Invariant.transformers_ok_mono_states

Terse proof for transformers_ok_mono_states.
transformers_ok_mono_states:

{1} ∀ (transformers: PRED[[State → SuperResult[State]]], states_1, states_2: PRED[State]):
(states_1 ⊆ states_2) ∧ transformers_ok?(states_2, transformers) ⊃
transformers_ok?(states_1, transformers)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of transformers_ok_mono_states.
Q.E.D.

C.229.11 Transformer_Invariant.transformers_ok_union_transformers

Terse proof for transformers_ok_union_transformers.
transformers_ok_union_transformers:

{1} ∀ (states: PRED[State],
transformers_1, transformers_2: PRED[[State → SuperResult[State]]]):
transformers_ok?(states, transformers_1) ∧ transformers_ok?(states, transformers_2) ⊃
transformers_ok?(states, (transformers_1 ∪ transformers_2))

Installing automatic rewrites from: member
Repeatedly Skolemizing and flattening,
Expanding the definition of transformers_ok?,

Repeatedly Skolemizing and flattening,
Expanding the definition of union,
Splitting conjunctions,
we get 2 subgoals:
`transformers_ok_union_transformers.1:`

$$
\begin{array}{ll}
\{\text{-}1\} & (q' \in \text{transformers}') \\
\{\text{-}2\} & \forall\ (s\colon \text{State},\ q\colon \lceil \text{State} \to \text{SuperResult}\lceil \text{State} \rceil \rceil)\colon \\
 & \quad \text{states}'(s)\ \wedge\ \text{transformers}'(q)\ \supset\ \text{OK?}(q(s)) \\
\{\text{-}3\} & \forall\ (s\colon \text{State},\ q\colon \lceil \text{State} \to \text{SuperResult}\lceil \text{State} \rceil \rceil)\colon \\
 & \quad \text{states}'(s)\ \wedge\ \text{transformers}''(q)\ \supset\ \text{OK?}(q(s)) \\
\{\text{-}4\} & \text{states}'(s') \\
\hline
\{1\} & \text{OK?}(q'(s'))
\end{array}
$$

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformers_ok_union_transformers.1`.
`transformers_ok_union_transformers.2:`

$$
\begin{array}{ll}
\{\text{-}1\} & (q' \in \text{transformers}'') \\
\{\text{-}2\} & \forall\ (s\colon \text{State},\ q\colon \lceil \text{State} \to \text{SuperResult}\lceil \text{State} \rceil \rceil)\colon \\
 & \quad \text{states}'(s)\ \wedge\ \text{transformers}'(q)\ \supset\ \text{OK?}(q(s)) \\
\{\text{-}3\} & \forall\ (s\colon \text{State},\ q\colon \lceil \text{State} \to \text{SuperResult}\lceil \text{State} \rceil \rceil)\colon \\
 & \quad \text{states}'(s)\ \wedge\ \text{transformers}''(q)\ \supset\ \text{OK?}(q(s)) \\
\{\text{-}4\} & \text{states}'(s') \\
\hline
\{1\} & \text{OK?}(q'(s'))
\end{array}
$$

Instantiating quantified variables,
Simplifying, rewriting, and recording with decision procedures,
This completes the proof of `transformers_ok_union_transformers.2`.
Q.E.D.

C.229.12 Transformer_Invariant.transformers_ok_all_transformers

Terse proof for `transformers_ok_all_transformers`.
`transformers_ok_all_transformers:`

$$
\begin{array}{ll}
\{1\} & \forall\ (\text{states}\colon \text{PRED}\lceil \text{State} \rceil,\ \text{transformers}\colon \text{PRED}\lceil \lceil \text{State} \to \text{SuperResult}\lceil \text{State} \rceil \rceil \rceil)\colon \\
 & \quad (\forall\ (q\colon \lceil \text{State} \to \text{SuperResult}\lceil \text{State} \rceil \rceil)\colon \\
 & \qquad \text{transformers}(q)\ \supset\ \text{transformers\_ok?}(\text{states},\ \text{singleton}(q))) \\
 & \quad \supset\ \text{transformers\_ok?}(\text{states},\ \text{transformers})
\end{array}
$$

Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating quantified variables,
Expanding the definition of singleton,
which is trivially true.
This completes the proof of `transformers_ok_all_transformers`.
Q.E.D.

C.230 Proofs for Transformer_Invariant_2 (state-transformer.pvs)

C.230.1 Transformer_Invariant_2.expr_transformer_invariant_next_ok_TCC1

Terse proof for `expr_transformer_invariant_next_ok_TCC1`.
`expr_transformer_invariant_next_ok_TCC1`:

{1} ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → ExprResult[State, Data]]):
transformer_invariant?(states, transformers) ∧
states(s) ∧ transformers(expr_2_super(q)) ∧ has_next_state($q(s)$)
⊃ OK?[State, Data]($q(s)$) ∨ Exception?[State, Data]($q(s)$)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_transformer_invariant_next_ok_TCC1`.
Q.E.D.

C.230.2 Transformer_Invariant_2.expr_transformer_invariant_next_ok

Terse proof for `expr_transformer_invariant_next_ok`.
`expr_transformer_invariant_next_ok`:

{1} ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → ExprResult[State, Data]]):
transformer_invariant?(states, transformers) ∧
states(s) ∧ transformers(expr_2_super(q)) ∧ has_next_state($q(s)$)
⊃ states(state($q(s)$))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_transformer_invariant_next_ok`.
Q.E.D.

C.230.3 Transformer_Invariant_2.expr_transformers_ok_ok

Terse proof for `expr_transformers_ok_ok`.
`expr_transformers_ok_ok`:

{1} ∀ (states: PRED[State], transformers: PRED[[State → SuperResult[State]]], s: State,
q: [State → ExprResult[State, Data]]):
transformers_ok?(states, transformers) ∧ states(s) ∧ transformers(expr_2_super(q)) ⊃
OK?($q(s)$)

Repeatedly Skolemizing and flattening,
Using lemma super_transformers_ok_ok[State],

Simplifying, rewriting, and recording with decision procedures,
Rewriting using ok_expr_2_super, matching in *,
This completes the proof of `expr_transformers_ok_ok`.
Q.E.D.

C.230.4 Transformer_Invariant_2.expr_result_pred_next_state_TCC1

Terse proof for `expr_result_pred_next_state_TCC1`.
`expr_result_pred_next_state_TCC1:`

{1} ∀ (states: PRED[State], s: State, q: [State → ExprResult[State, Data]]):
has_next_state($q(s)$) ∧ result_pred(states)(expr_2_super(q)(s)) ⊃
OK?[State, Data]($q(s)$) ∨ Exception?[State, Data]($q(s)$)

Repeatedly Skolemizing and flattening,
Expanding the definition of has_next_state,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `expr_result_pred_next_state_TCC1`.
Q.E.D.

C.230.5 Transformer_Invariant_2.expr_result_pred_next_state

Terse proof for `expr_result_pred_next_state`.
`expr_result_pred_next_state:`

{1} ∀ (states: PRED[State], s: State, q: [State → ExprResult[State, Data]]):
has_next_state($q(s)$) ∧ result_pred(states)(expr_2_super(q)(s)) ⊃
states(state($q(s)$))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `expr_result_pred_next_state`.
Q.E.D.

C.231 Proofs for Transformer_Invariant_3 (state-transformer.pvs)

C.231.1 Transformer_Invariant_3.fexpr_composition_transformers_ok

Terse proof for `fexpr_composition_transformers_ok`.
`fexpr_composition_transformers_ok:`

{1} ∀ (expr1: [State → ExprResult[State, Data1]],
fexpr: [Data1 → [State → ExprResult[State, Data2]]], states: PRED[State],
P: PRED[Data1]):
(transformers_ok?(states, singleton(expr_2_super(expr1))) ∧
transformer_invariant?(states, singleton(expr_2_super(expr1))) ∧
(∀ (s: (states)): OK?(expr1(s)) ⊃ P(data(expr1(s)))) ∧
(∀ (d: (P)):
transformers_ok?(states, singleton(expr_2_super(fexpr(d))))))
⊃ transformers_ok?(states, singleton(expr_2_super(expr1 ## fexpr)))

Expanding the definition of transformers_ok?,
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: singleton ## expr_2_super expr_2_super_res
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Replacing using formula -6,
Hiding formulas: -6,
Instantiating quantified variables,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -5 with the terms: (data(expr1!1(s!1))),
Instantiating the top quantifier in -5 with the terms: (state(expr1!1(s!1)) expr_2_super(fexpr!1(data(expr1!1(s!1))))),
we get 2 subgoals:
`fexpr_composition_transformers_ok.1:`

{-1} $\text{states}'(s')$
{-2} $\text{OK?}(\text{expr\_2\_super\_res}(\text{expr1}'(s')))$
{-3} $\text{transformer\_invariant?}(\text{states}', \text{singleton}(\text{expr\_2\_super}(\text{expr1}')))$
{-4} $P'(\text{data}(\text{expr1}'(s')))$
{-5} $\text{states}'(\text{state}(\text{expr1}'(s'))) \supset \text{OK?}(\text{expr\_2\_super}(\text{fexpr}'(\text{data}(\text{expr1}'(s'))))(\text{state}(\text{expr1}'(s'))))$

{1} $\text{OK?}(\text{expr\_2\_super\_res}((\text{expr1}' \ \#\# \ \text{fexpr}')(s')))$

Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma expr_transformer_invariant_next_ok,
Installing automatic rewrites from: has_next_state
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `fexpr_composition_transformers_ok.1`.
`fexpr_composition_transformers_ok.2:`

{-1} $\text{states}'(s')$
{-2} $\text{OK?}(\text{expr\_2\_super\_res}(\text{expr1}'(s')))$
{-3} $\text{transformer\_invariant?}(\text{states}', \text{singleton}(\text{expr\_2\_super}(\text{expr1}')))$
{-4} $P'(\text{data}(\text{expr1}'(s')))$

{1} $\text{OK?}[\text{State}, \text{Data1}](\text{expr1}'(s')) \vee \text{Exception?}[\text{State}, \text{Data1}](\text{expr1}'(s'))$
{2} $\text{OK?}(\text{expr\_2\_super\_res}((\text{expr1}' \ \#\# \ \text{fexpr}')(s')))$

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `fexpr_composition_transformers_ok.2`.
Q.E.D.

C.231.2 Transformer_Invariant_3.expr_composition_transformers_ok

Terse proof for `expr_composition_transformers_ok`.
`expr_composition_transformers_ok:`

{1} $\forall$ (expr1: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data1}]]$,
expr2: $[\text{State} \to \text{ExprResult}[\text{State}, \text{Data2}]]$, states: $\text{PRED}[\text{State}]$):
$\text{transformers\_ok?}(\text{states}, \text{singleton}(\text{expr\_2\_super}(\text{expr1}))) \wedge$
$\text{transformer\_invariant?}(\text{states}, \text{singleton}(\text{expr\_2\_super}(\text{expr1}))) \wedge$
$\text{transformers\_ok?}(\text{states}, \text{singleton}(\text{expr\_2\_super}(\text{expr2})))$
$\supset \text{transformers\_ok?}(\text{states}, \text{singleton}(\text{expr\_2\_super}(\text{expr1} \ \#\# \ \text{expr2})))$

Repeatedly Skolemizing and flattening,

Using lemma fexpr_composition_transformers_ok,
Expanding the definition of ##,
Expanding the definition of ##,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `expr_composition_transformers_ok`.
Q.E.D.

C.231.3 Transformer_Invariant_3.fexpr_composition_transformer_invariant

Terse proof for `fexpr_composition_transformer_invariant`.
`fexpr_composition_transformer_invariant:`

```
{1}  ∀ (expr1: [State → ExprResult[State, Data1]],
          fexpr: [Data1 → [State → ExprResult[State, Data2]]], states: PRED[State],
          P: PRED[Data1]):
       (transformer_invariant?(states, singleton(expr_2_super(expr1))) ∧
          (∀ (s: (states)): OK?(expr1(s)) ⊃ P(data(expr1(s)))) ∧
           (∀ (d: (P)):
                transformer_invariant?(states, singleton(expr_2_super(fexpr(d))))))
        ⊃ transformer_invariant?(states, singleton(expr_2_super(expr1 ## fexpr)))
```

Expanding the definition of transformer_invariant?,
Repeatedly Skolemizing and flattening,
Expanding the definition of singleton,
Replacing using formula -5,
Hiding formulas: -5,
Instantiating quantified variables,
Instantiating quantified variables,
Case splitting on OK?(expr1!1(s!1)),
we get 2 subgoals:
`fexpr_composition_transformer_invariant.1:`

```
{-1}  OK?(expr1′(s′))
{-2}  states′(s′) ⊃ result_pred(states′)(expr_2_super(expr1′)(s′))
{-3}  OK?(expr1′(s′)) ⊃ P′(data(expr1′(s′)))
{-4}  ∀ (d: (P′)):
        ∀ (s: State, q: [State → SuperResult[State]]):
          states′(s) ∧ q = expr_2_super(fexpr′(d)) ⊃ result_pred(states′)(q(s))
{-5}  states′(s′)
------------------------------------------------------------------------------
 {1}  result_pred(states′)(expr_2_super(expr1′ ## fexpr′)(s′))
```

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -5 with the terms: (data(expr1!1(s!1))),
Instantiating the top quantifier in -5 with the terms: (state(expr1!1(s!1)) expr_2_super(fexpr!1(data(expr1!1(s!1))),
Installing automatic rewrites from: ## result_pred expr_2_super expr_2_super_res has_next_state
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `fexpr_composition_transformer_invariant.1`.

`fexpr_composition_transformer_invariant.2`:

{-1} $\text{states}'(s') \supset \text{result\_pred}(\text{states}')(\text{expr\_2\_super}(\text{expr1}')(s'))$
{-2} $\text{OK?}(\text{expr1}'(s')) \supset P'(\text{data}(\text{expr1}'(s')))$
{-3} $\forall\ (d\colon (P')):$
$\quad \forall\ (s\colon \text{State},\ q\colon [\text{State} \to \text{SuperResult}[\text{State}]]):$
$\quad\quad \text{states}'(s) \wedge q = \text{expr\_2\_super}(\text{fexpr}'(d)) \supset \text{result\_pred}(\text{states}')(q(s))$
{-4} $\text{states}'(s')$

{1} $\text{OK?}(\text{expr1}'(s'))$
{2} $\text{result\_pred}(\text{states}')(\text{expr\_2\_super}(\text{expr1}' \ \#\#\ \text{fexpr}')(s'))$

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `fexpr_composition_transformer_invariant.2`.
Q.E.D.

C.231.4 Transformer_Invariant_3.expr_composition_transformer_invariant

Terse proof for `expr_composition_transformer_invariant`.

`expr_composition_transformer_invariant`:

{1} $\forall\ (\text{expr1}\colon [\text{State} \to \text{ExprResult}[\text{State},\ \text{Data1}]],$
$\quad\quad \text{expr2}\colon [\text{State} \to \text{ExprResult}[\text{State},\ \text{Data2}]],\ \text{states}\colon \text{PRED}[\text{State}]):$
$\quad \text{transformer\_invariant?}(\text{states},\ \text{singleton}(\text{expr\_2\_super}(\text{expr1}))) \wedge$
$\quad\ \text{transformer\_invariant?}(\text{states},\ \text{singleton}(\text{expr\_2\_super}(\text{expr2})))$
$\quad \supset \text{transformer\_invariant?}(\text{states},\ \text{singleton}(\text{expr\_2\_super}(\text{expr1} \ \#\#\ \text{expr2})))$

Repeatedly Skolemizing and flattening,
Using lemma fexpr_composition_transformer_invariant,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of ##,
Expanding the definition of ##,
which is trivially true.
This completes the proof of `expr_composition_transformer_invariant`.
Q.E.D.

C.232 Proofs for Transformer_Super_Embedding (state-transformer.pvs)

This theory contains no provable formal statements.

C.233 Proofs for Transformer_Super_Embedding_Expr (state-transformer.pvs)

C.233.1 Transformer_Super_Embedding_Expr.ok_expr_2_super

Terse proof for `ok_expr_2_super`.

ok_expr_2_super:

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State):
OK?(expr_2_super(expr)(s)) = OK?(expr(s))

,
This completes the proof of ok_expr_2_super.
Q.E.D.

C.233.2 Transformer_Super_Embedding_Expr.has_next_state_expr_2_super

Terse proof for has_next_state_expr_2_super.
has_next_state_expr_2_super:

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State):
has_next_state(expr_2_super(expr)(s)) = has_next_state(expr(s))

,
This completes the proof of has_next_state_expr_2_super.
Q.E.D.

C.233.3 Transformer_Super_Embedding_Expr.state_expr_2_super_TCC1

Terse proof for state_expr_2_super_TCC1.
state_expr_2_super_TCC1:

{1} ∀ (expr: [State → ExprResult[State, Data]], s: State):
has_next_state(expr(s)) ⊃
OK?[State](expr_2_super(expr)(s)) ∨ abnormal?[State](expr_2_super(expr)(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of state_expr_2_super_TCC1.
Q.E.D.

C.233.4 Transformer_Super_Embedding_Expr.state_expr_2_super_TCC2

Terse proof for state_expr_2_super_TCC2.
state_expr_2_super_TCC2:

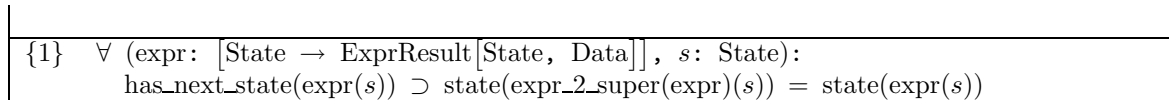
{1} ∀ (expr: [State → ExprResult[State, Data]], s: State):
has_next_state(expr(s)) ⊃
OK?[State, Data](expr(s)) ∨ Exception?[State, Data](expr(s))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of state_expr_2_super_TCC2.
Q.E.D.

C.233.5 Transformer_Super_Embedding_Expr.state_expr_2_super

Terse proof for `state_expr_2_super`.

`state_expr_2_super:`



,

This completes the proof of `state_expr_2_super`.

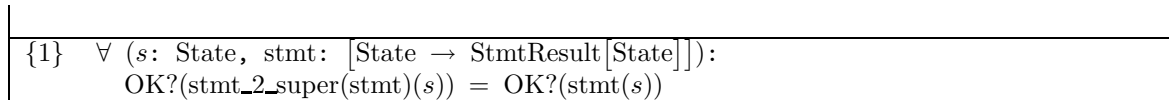
Q.E.D.

C.234 Proofs for Transformer_Super_Embedding_Stmt (state-transformer.pvs)

C.234.1 Transformer_Super_Embedding_Stmt.ok_stmt_2_super

Terse proof for `ok_stmt_2_super`.

`ok_stmt_2_super:`



,

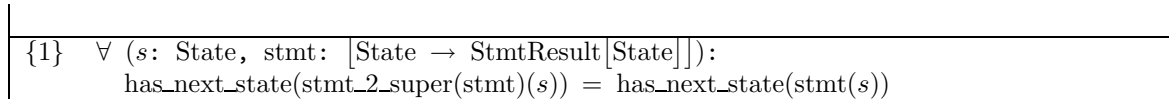
This completes the proof of `ok_stmt_2_super`.

Q.E.D.

C.234.2 Transformer_Super_Embedding_Stmt.has_next_state_stmt_2_super

Terse proof for `has_next_state_stmt_2_super`.

`has_next_state_stmt_2_super:`



,

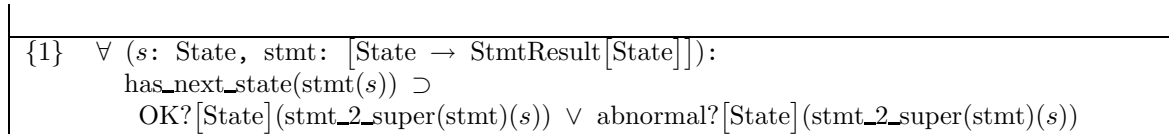
This completes the proof of `has_next_state_stmt_2_super`.

Q.E.D.

C.234.3 Transformer_Super_Embedding_Stmt.state_stmt_2_super_TCC1

Terse proof for `state_stmt_2_super_TCC1`.

`state_stmt_2_super_TCC1:`

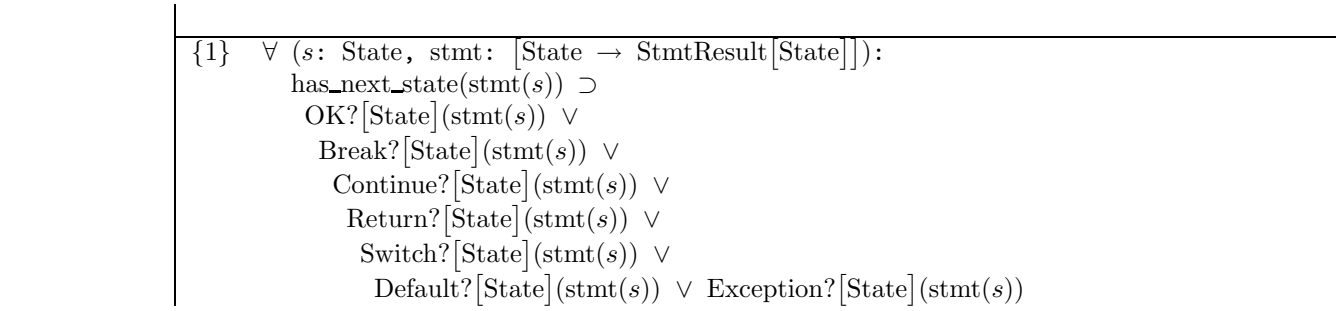


Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `state_stmt_2_super_TCC1`.
Q.E.D.

C.234.4 Transformer_Super_Embedding_Stmt.state_stmt_2_super_TCC2

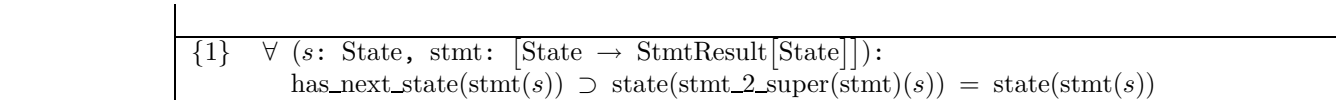
Terse proof for `state_stmt_2_super_TCC2`.
`state_stmt_2_super_TCC2:`



Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `state_stmt_2_super_TCC2`.
Q.E.D.

C.234.5 Transformer_Super_Embedding_Stmt.state_stmt_2_super

Terse proof for `state_stmt_2_super`.
`state_stmt_2_super:`

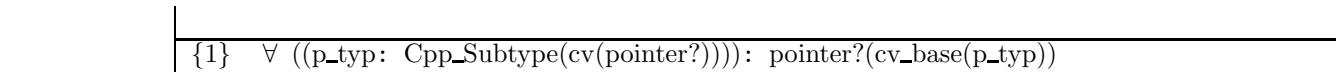


,
This completes the proof of `state_stmt_2_super`.
Q.E.D.

C.235 Proofs for UnaryExpressions (expressions.pvs)

C.235.1 UnaryExpressions.deref_TCC1

Terse proof for `deref_TCC1`.
`deref_TCC1:`



Repeatedly Skolemizing and flattening,
Rewriting using cv_base_result, matching in *,
Keeping (1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `deref_TCC1`.
Q.E.D.

C.235.2 UnaryExpressions.arrow_TCC1

Terse proof for `arrow_TCC1`.

`arrow_TCC1`:

{1} ∀ (t: Cpp_Type):
pointer?(t) ⊃
array?(t) ∨
pointer?(t) ∨
reference?(t) ∨
bitfield?(t) ∨ enum?(t) ∨ pointer_to_member?(t) ∨ const?(t) ∨ volatile?(t)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `arrow_TCC1`.
Q.E.D.

C.235.3 UnaryExpressions.arrow_TCC2

Terse proof for `arrow_TCC2`.

`arrow_TCC2`:

{1} ∀ ((p_typ:
Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
(λ (t: Cpp_Type):
pointer?(t) ∧
(class?(typ(t)) ∨ union?(typ(t))))))):
cv(pointer?)(p_typ)

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `arrow_TCC2`.
Q.E.D.

C.235.4 UnaryExpressions.unary_minus_TCC1

Terse proof for `unary_minus_TCC1`.

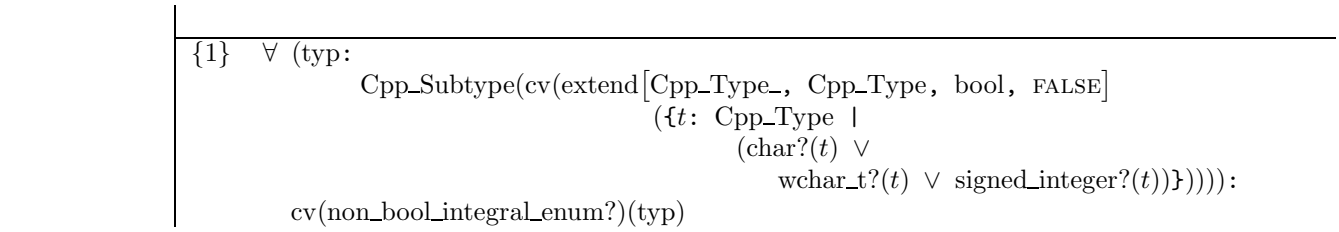
`unary_minus_TCC1`:

{1} ∀ (typ:
Cpp_Subtype(cv(extend[Cpp_Type_, Cpp_Type, bool, FALSE]
({t: Cpp_Type |
(char?(t) ∨
wchar_t?(t) ∨ signed_integer?(t))})),
n: (range(typ))):
cv(non_bool_integral_enum?)(typ)

Repeatedly Skolemizing and flattening,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unary_minus_TCC1`.
Q.E.D.

C.235.5 UnaryExpressions.unary_minus_TCC2

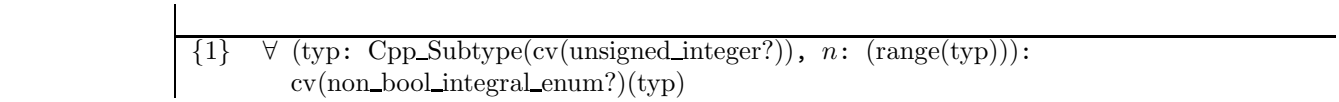
Terse proof for `unary_minus_TCC2`.
`unary_minus_TCC2:`



Repeatedly Skolemizing and flattening,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unary_minus_TCC2`.
Q.E.D.

C.235.6 UnaryExpressions.unary_minus_unsigned_TCC1

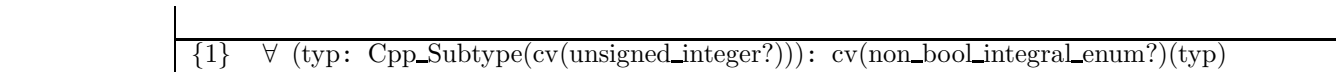
Terse proof for `unary_minus_unsigned_TCC1`.
`unary_minus_unsigned_TCC1:`



Repeatedly Skolemizing and flattening,
Keeping (-2 1) and hiding *,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unary_minus_unsigned_TCC1`.
Q.E.D.

C.235.7 UnaryExpressions.unary_minus_unsigned_TCC2

Terse proof for `unary_minus_unsigned_TCC2`.
`unary_minus_unsigned_TCC2:`



Repeatedly Skolemizing and flattening,
Hiding formulas: -1,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `unary_minus_unsigned_TCC2`.
Q.E.D.

C.235.8 UnaryExpressions.preinc_TCC1

Terse proof for `preinc_TCC1`.
`preinc_TCC1:`

Repeatedly Skolemizing and flattening,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `preinc_TCC1`.
Q.E.D.

C.235.9 UnaryExpressions.sizeof_TCC1

Terse proof for `sizeof_TCC1`.
`sizeof_TCC1`:

{1} Cpp_Type?(uint) ∧ cv(non_bool_integral_enum?)(uint)

Trying repeated skolemization, instantiation, and if-lifting,
Expanding the definition of Cpp_Type?,
Repeatedly Skolemizing and flattening,
Expanding the definition of subterm,
Replacing using formula -1,
Installing automatic rewrites from: no_pointers_to_bitfield? no_pointers_to_references? no_cv_references? no_reference_to_reference? no_reference_to_bitfields? no_pointer_to_member_to_reference? no_pointer_to_member_to_cv_void? no_cv_void_parameter? no_array_of_references? no_array_of_bitfields? no_array_of_cv_void? no_array_of_function? no_array_of_abstract_class? no_pointer_or_ref_to_incomplete_array_parameter? no_array_return_type? no_function_return_type? bitfield_underlying_integral_or_enum_type? cv_array? enum_underlying_integral? enum_constants? const_volatile? const_stutter? volatile_stutter? no_cv_class? no_cv_union? no_cv_function? no_reference_to_void? no_cv_void?
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `sizeof_TCC1`.
Q.E.D.

C.236 Proofs for Uninterpreted_Data (abstract_data.pvs)

C.236.1 Uninterpreted_Data.u_data_type_length

Terse proof for `u_data_type_length`.
`u_data_type_length`:

{1} ∀ (uidt: (uninterpreted_data_type?), l: list⌊Byte⌋, a: Address):
valid?(uidt)(l, a) ⊃ length(l) = size(uidt)

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `u_data_type_length`.
Q.E.D.

C.237 Proofs for Unit (vfiasco-prelude.pvs)

This theory contains no provable formal statements.

C.238 Proofs for Unit_adt (Unit_adt.pvs)

This theory contains no provable formal statements.

C.239 Proofs for Unit_adt_reduce (Unit_adt.pvs)

This theory contains no provable formal statements.

C.240 Proofs for While_Hoare (statement-rewrites.pvs)

C.240.1 While_Hoare.iterate_while_right

Terse proof for `iterate_while_right`.

`iterate_while_right:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          n: nat):
       iterate_while(1 + n, b_ex, body) =
        iterate_while(n, b_ex, body) ## lift(e2s(b_ex) ## body ## catch_continue)
```

Installing automatic rewrites from: erplus_ax erminus_ax extended_real_superset_of_number_fields
Inducting on n on formula 1,
we get 2 subgoals:

`iterate_while_right.1:`

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]]):
       iterate_while(1 + 0, b_ex, body) =
        iterate_while(0, b_ex, body) ## lift(e2s(b_ex) ## body ## catch_continue)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Trying repeated skolemization, instantiation, and if-lifting,
Applying eta axiom scheme to body!1(state(b_ex!1(x!1))),
This completes the proof of `iterate_while_right.1`.

`iterate_while_right.2:`

```
{1}  ∀ j:
       (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
     sult[State]]):
          iterate_while(1 + j, b_ex, body) =
           iterate_while(j, b_ex, body) ## lift(e2s(b_ex) ## body ## catch_continue))
       ⊃
       (∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtRe-
     sult[State]]):
          iterate_while(1 + j + 1, b_ex, body) =
           iterate_while(j + 1, b_ex, body) ##
            lift(e2s(b_ex) ## body ## catch_continue))
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,

Expanding the definition of iterate_while,

Expanding the definition of ##,

Applying decompose-equality,

Expanding the definition of lift,

Replacing using formula -2,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `iterate_while_right.2`.

Q.E.D.

C.240.2 While_Hoare.while_variant?_TCC1

Terse proof for `while_variant?_TCC1`.

while_variant?_TCC1:

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
<: PRED[[Ord, Ord]], variant: [State → Ord], Q: PRED[StmtResult[State]]):
while_invariant?(b_ex, body)(Q, Q) ∧ well_founded?(<) ⊃
(∀ (s: State):
Q(OK(s)) ∧
OK?(b_ex(s)) ∧
data(b_ex(s)) ∧ OK?((e2s(b_ex) ## body ## catch_continue)(s))
⊃
OK?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s))
∨
Break?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s))
∨
Continue?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s))
∨
Return?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s))
∨
Switch?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s))
∨
Default?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s))
∨
Exception?[State]
((##[State]
(##[State](e2s[State, boolean](b_ex), body),
catch_continue[State]))
(s)))

Repeatedly Skolemizing and flattening,
This completes the proof of `while_variant?_TCC1`.
Q.E.D.

C.240.3 While_Hoare.while_no_cb_TCC1

Terse proof for `while_no_cb_TCC1`.

`while_no_cb_TCC1`:

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          s: State):
       (∃ (n: nat): while_termination_point?(b_ex, body, s)(n)) ⊃
        nonempty?[nat](while_termination_point?[State](b_ex, body, s))
```

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_no_cb_TCC1`.
Q.E.D.

C.240.4 While_Hoare.while_no_cb_cb_while

Terse proof for `while_no_cb_cb_while`.

`while_no_cb_cb_while`:

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]]):
       while(b_ex, body) = (while_no_cb(b_ex, body) ## catch_break)
```

Repeatedly Skolemizing and flattening,
Applying decompose-equality,
Expanding the definition of while,
Expanding the definition of while_no_cb,
Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `while_no_cb_cb_while`.
Q.E.D.

C.240.5 While_Hoare.iterate_while_invariant

Terse proof for `iterate_while_invariant`.

`iterate_while_invariant`:

```
{1}  ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]], n: nat):
       (while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) ∧
          inv_abnorm?(OK(s_0)) ∧
           (∀ (m: nat): m < n ⊃ ¬ while_termination_point?(b_ex, body, s_0)(m))
        ⊃ inv_abnorm?(iterate_while(n, b_ex, body)(s_0))
```

Inducting on n on formula 1,
we get 2 subgoals:

`iterate_while_invariant.1:`

{1} ∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]]):
(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) ∧
inv_abnorm?(OK(s_0)) ∧
(∀ (m: nat): $m < 0$ ⊃ ¬ while_termination_point?(b_ex, body, s_0)(m)))
⊃ inv_abnorm?(iterate_while(0, b_ex, body)(s_0))

Trying repeated skolemization, instantiation, and if-lifting,
This completes the proof of `iterate_while_invariant.1`.
`iterate_while_invariant.2:`

{1} ∀ j:
(∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]]):
(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) ∧
inv_abnorm?(OK(s_0)) ∧
(∀ (m: nat): $m < j$ ⊃ ¬ while_termination_point?(b_ex, body, s_0)(m)))
⊃ inv_abnorm?(iterate_while(j, b_ex, body)(s_0)))
⊃
(∀ (b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
s_0: State, inv_abnorm?, inv_norm?: PRED[StmtResult[State]]):
(while_invariant?(b_ex, body)(inv_abnorm?, inv_norm?) ∧
inv_abnorm?(OK(s_0)) ∧
(∀ (m: nat):
$m < j + 1$ ⊃ ¬ while_termination_point?(b_ex, body, s_0)(m)))
⊃ inv_abnorm?(iterate_while($j + 1$, b_ex, body)(s_0)))

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Instantiating the top quantifier in -2 with the terms: (s0!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
we get 2 subgoals:
`iterate_while_invariant.2.1:`

{-1} $j' \geq 0$
{-2} while_invariant?(b_ex′, body′)(inv_abnorm?′, inv_norm?′)
{-3} inv_abnorm?′(OK(s_0'))
{-4} inv_abnorm?′(iterate_while(j', b_ex′, body′)(s_0'))
{-5} ∀ (m: nat): $m < 1 + j' \supset \neg$ while_termination_point?(b_ex′, body′, s_0')(m)
{1} inv_abnorm?′(iterate_while($1 + j'$, b_ex′, body′)(s_0'))

Expanding the definition of while_invariant?,
Using lemma iterate_while_right,
Simplifying, rewriting, and recording with decision procedures,
Replacing using formula -1,
Instantiating the top quantifier in -6 with the terms: (j!1),
Expanding the definition of while_termination_point?,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Instantiating the top quantifier in -3 with the terms: (state(iterate_while(j!1, b_ex!1, body!1)(s0!1))),

Applying eta axiom scheme to iterate_while(j!1, b_ex!1, body!1)(s0!1),
Expanding the definition of ##,
Expanding the definition of lift,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of skip,
Replacing using formula -1,
which is trivially true.
This completes the proof of `iterate_while_invariant.2.1`.
`iterate_while_invariant.2.2`:

```
{-1}  j' >= 0
{-2}  while_invariant?(b_ex', body')(inv_abnorm?', inv_norm?')
{-3}  inv_abnorm?'(OK(s0'))
{-4}  FORALL (m: nat): m < 1 + j' IMPLIES NOT while_termination_point?(b_ex', body', s0')(m)
------------------------------------------------------------------------------------------
{1}   FORALL (m: nat): m < j' IMPLIES NOT while_termination_point?(b_ex', body', s0')(m)
{2}   inv_abnorm?'(iterate_while(1 + j', b_ex', body')(s0'))
```

Repeatedly Skolemizing and flattening,
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `iterate_while_invariant.2.2`.
Q.E.D.

C.240.6 While_Hoare.min_termination_point_no_less

Terse proof for `min_termination_point_no_less`.
`min_termination_point_no_less`:

```
{1}  FORALL (b_ex: [State -> ExprResult[State, bool]], body: [State -> StmtResult[State]],
             s0: State, m: nat):
       (EXISTS (n: nat): while_termination_point?(b_ex, body, s0)(n)) AND
        m < min[nat](while_termination_point?(b_ex, body, s0))
        IMPLIES NOT while_termination_point?(b_ex, body, s0)(m)
```

Repeatedly Skolemizing and flattening,
Adding type constraints for min[nat](while_termination_point?(b_ex!1, body!1, s0!1)),
Instantiating quantified variables,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `min_termination_point_no_less`.
Q.E.D.

C.240.7 While_Hoare.while_terminates

The LaTeX code for this proof is broken.

C.240.8 While_Hoare.hoare_while

The LaTeX code for this proof is broken.

C.241 Proofs for While_Rewrites (statement-rewrites.pvs)

C.241.1 While_Rewrites.while_to_while_no_cb

Terse proof for `while_to_while_no_cb`.

`while_to_while_no_cb`:

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
          variant: [State → [State → Ord]]):
       while[State, Ord](b_ex, body)(P, Q, R, variant, <) =
         (while_no_cb(b_ex, body)(P, Q, R, variant, <) ## catch_break)
```

Expanding the definition of while_no_cb,
Expanding the definition of while,
Repeatedly Skolemizing and flattening,
Using lemma while_no_cb_cb_while[State, Ord],
This completes the proof of `while_to_while_no_cb`.
Q.E.D.

C.241.2 While_Rewrites.while_inv_rewrite_termination_result

Terse proof for `while_inv_rewrite_termination_result`.

`while_inv_rewrite_termination_result`:

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          S: PRED[StmtResult[State]], b_ex: [State → ExprResult[State, bool]],
          body: [State → StmtResult[State]], s: State, variant: [State → [State → Ord]]):
       (P(s)(skip(s)) ∧
          R(s)(skip(s)) ∧
            while_variant?(b_ex, body, <)(variant(s), R(s)) ∧
              while_invariant?(b_ex, body)(P(s), Q(s)) ∧
                (∀ (res: StmtResult[State]): R(s)(res) ⊃ S(res)))
         ⊃ S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
```

Repeatedly Skolemizing and flattening,
Using lemma hoare_while[State, Ord],
Installing automatic rewrites from: skip while_no_cb
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of `while_inv_rewrite_termination_result`.
Q.E.D.

C.241.3 While_Rewrites.while_inv_rewrite_data_ok

Terse proof for `while_inv_rewrite_data_ok`.

`while_inv_rewrite_data_ok`:

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          S: PRED[StmtResult[State]], b_ex: [State → ExprResult[State, bool]],
          body: [State → StmtResult[State]], s: State, vari-
      ant: [State → [State → Ord]]):
        (P(s)(skip(s)) ∧
           R(s)(skip(s)) ∧
            while_variant?(b_ex, body, <)(variant(s), R(s)) ∧
             while_invariant?(b_ex, body)(P(s), Q(s)) ∧
              OK?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s)) ∧
               (∀ (res: StmtResult[State]): Q(s)(res) ⊃ S(res)))
          ⊃ S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
```

Installing automatic rewrites from: skip while_no_cb
Repeatedly Skolemizing and flattening,
Using lemma hoare_while[State, Ord],
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of `while_inv_rewrite_data_ok`.
Q.E.D.

C.241.4 While_Rewrites.while_inv_rewrite_data_break

Terse proof for `while_inv_rewrite_data_break`.

`while_inv_rewrite_data_break`:

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          S: PRED[StmtResult[State]], b_ex: [State → ExprResult[State, bool]],
          body: [State → StmtResult[State]], s: State, vari-
      ant: [State → [State → Ord]]):
        (P(s)(skip(s)) ∧
           R(s)(skip(s)) ∧
            while_variant?(b_ex, body, <)(variant(s), R(s)) ∧
             while_invariant?(b_ex, body)(P(s), Q(s)) ∧
              Break?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s)) ∧
               (∀ (res: StmtResult[State]):
                    (P(s)(res) ∨ Q(s)(res)) ∧ Break?(res) ⊃ S(res)))
          ⊃ S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
```

Installing automatic rewrites from: skip while_no_cb
Repeatedly Skolemizing and flattening,
Using lemma hoare_while[State, Ord],
Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,
This completes the proof of `while_inv_rewrite_data_break`.
Q.E.D.

C.241.5 While_Rewrites.while_inv_rewrite_data_return

Terse proof for `while_inv_rewrite_data_return`.

`while_inv_rewrite_data_return:`

```
{1}  ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
          S: PRED[StmtResult[State]], b_ex: [State → ExprResult[State, bool]],
          body: [State → StmtResult[State]], s: State, vari-
     ant: [State → [State → Ord]]):
       (P(s)(skip(s)) ∧
          R(s)(skip(s)) ∧
           while_variant?(b_ex, body, <)(variant(s), R(s)) ∧
            while_invariant?(b_ex, body)(P(s), Q(s)) ∧
             Return?(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s)) ∧
              (∀ (res: StmtResult[State]):
                   (P(s)(res) ∨ Q(s)(res)) ∧ Return?(res) ⊃ S(res)))
         ⊃ S(while_no_cb(b_ex, body)(P, Q, R, variant, <)(s))
```

Installing automatic rewrites from: skip while_no_cb

Repeatedly Skolemizing and flattening,

Using lemma hoare_while[State, Ord],

Repeatedly simplifying with decision procedures, rewriting, propositional reasoning, quantifier instantiation, skolemization, if-lifting and equality replacement,

This completes the proof of `while_inv_rewrite_data_return`.

Q.E.D.

C.241.6 While_Rewrites.stmt_while_inv_rewrite_termination_result_TCC1

Terse proof for `stmt_while_inv_rewrite_termination_result_TCC1`.

`stmt_while_inv_rewrite_termination_result_TCC1:`

```
{1}  ∀ (s: State, stmt: [State → StmtResult[State]]):
       OK?(stmt(s)) ⊃
        OK?[State](stmt(s)) ∨
         Break?[State](stmt(s)) ∨
          Continue?[State](stmt(s)) ∨
           Return?[State](stmt(s)) ∨
            Switch?[State](stmt(s)) ∨
             Default?[State](stmt(s)) ∨ Exception?[State](stmt(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

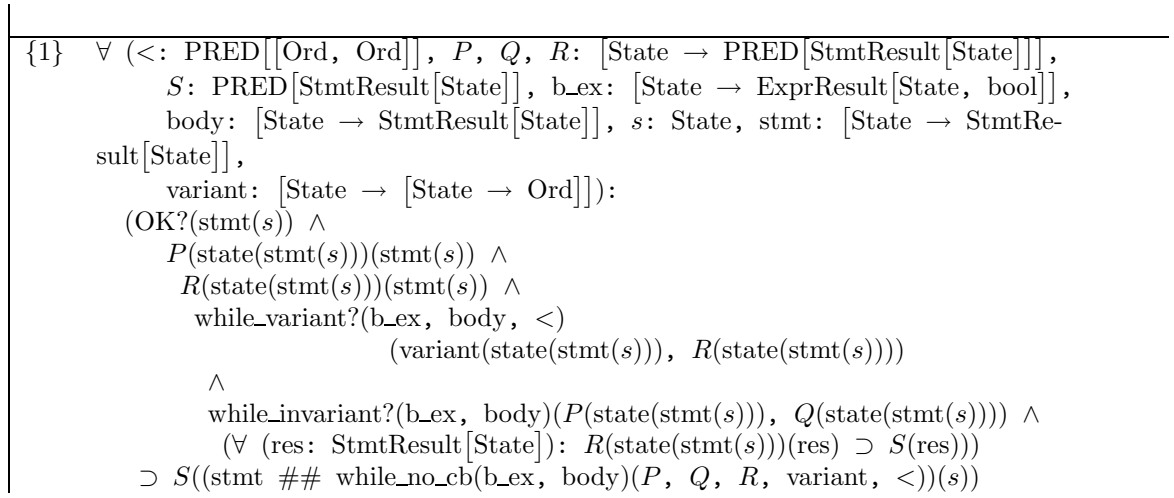
This completes the proof of `stmt_while_inv_rewrite_termination_result_TCC1`.

Q.E.D.

C.241.7 While_Rewrites.stmt_while_inv_rewrite_termination_result

Terse proof for `stmt_while_inv_rewrite_termination_result`.

`stmt_while_inv_rewrite_termination_result`:

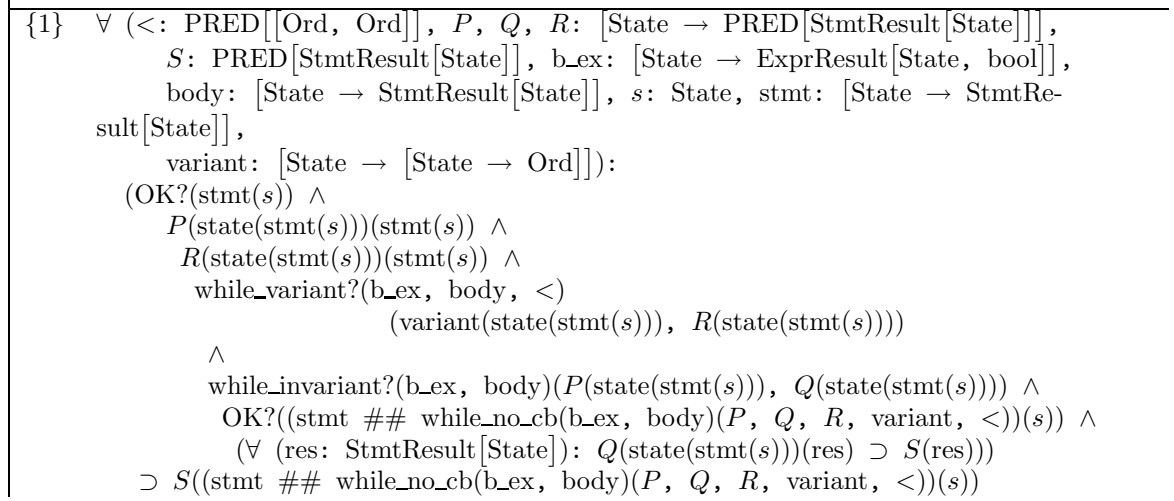


Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: ## skip lift!
Using lemma while_inv_rewrite_termination_result,
Applying eta axiom scheme to stmt!1(s!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `stmt_while_inv_rewrite_termination_result`.
Q.E.D.

C.241.8 While_Rewrites.stmt_while_inv_rewrite_data_ok

Terse proof for `stmt_while_inv_rewrite_data_ok`.

`stmt_while_inv_rewrite_data_ok`:



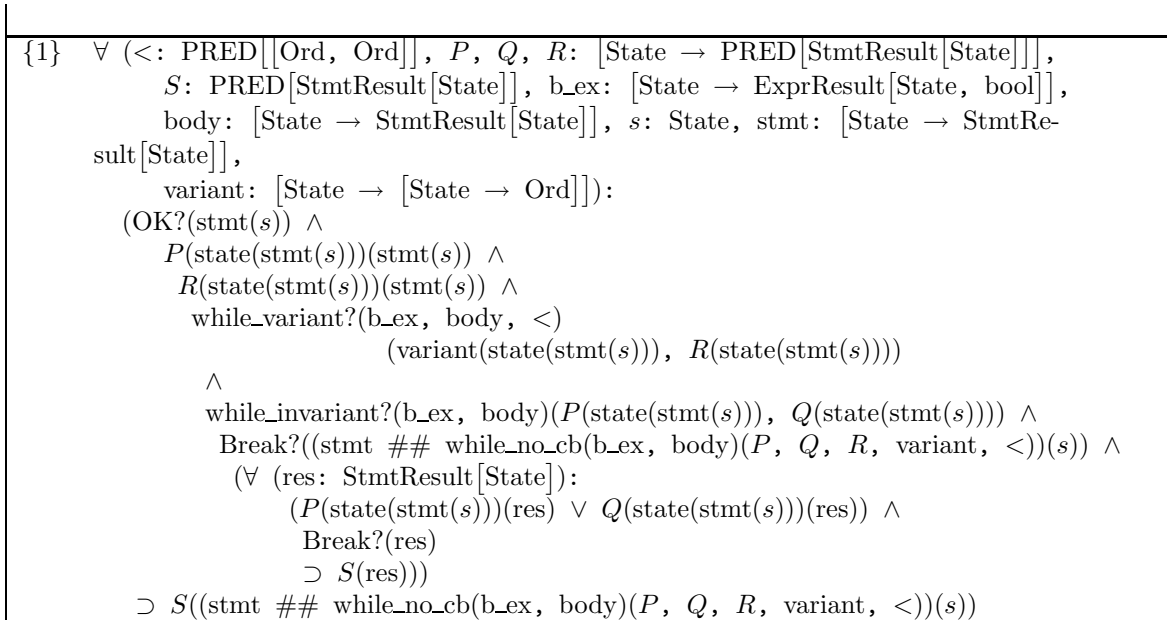
Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: ## skip lift!
Applying eta axiom scheme to stmt!1(s!1),
Using lemma while_inv_rewrite_data_ok,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `stmt_while_inv_rewrite_data_ok`.

Q.E.D.

C.241.9 While_Rewrites.stmt_while_inv_rewrite_data_break

Terse proof for `stmt_while_inv_rewrite_data_break`.

`stmt_while_inv_rewrite_data_break:`



Repeatedly Skolemizing and flattening,

Installing automatic rewrites from: ## skip lift!

Applying eta axiom scheme to stmt!1(s!1),

Using lemma while_inv_rewrite_data_break,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

This completes the proof of `stmt_while_inv_rewrite_data_break`.

Q.E.D.

C.241.10 While_Rewrites.stmt_while_inv_rewrite_data_return

Terse proof for `stmt_while_inv_rewrite_data_return`.

`stmt_while_inv_rewrite_data_return`:

{1} ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
S: PRED[StmtResult[State]], b_ex: [State → ExprResult[State, bool]],
body: [State → StmtResult[State]], s: State, stmt: [State → StmtResult[State]],
variant: [State → [State → Ord]]):
(OK?(stmt(s)) ∧
P(state(stmt(s)))(stmt(s)) ∧
R(state(stmt(s)))(stmt(s)) ∧
while_variant?(b_ex, body, <)
(variant(state(stmt(s))), R(state(stmt(s))))
∧
while_invariant?(b_ex, body)(P(state(stmt(s))), Q(state(stmt(s)))) ∧
Return?((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s)) ∧
(∀ (res: StmtResult[State]):
(P(state(stmt(s)))(res) ∨ Q(state(stmt(s)))(res)) ∧
Return?(res)
⊃ S(res)))
⊃ S((stmt ## while_no_cb(b_ex, body)(P, Q, R, variant, <))(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: ## skip lift!
Applying eta axiom scheme to stmt!1(s!1),
Using lemma while_inv_rewrite_data_return,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
This completes the proof of `stmt_while_inv_rewrite_data_return`.
Q.E.D.

C.241.11 While_Rewrites.pm_q_prop_while

Terse proof for `pm_q_prop_while`.

`pm_q_prop_while`:

{1} ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]],
b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]],
pm: Plain_Memory[State], s: State, variant: [State → [State → Ord]]):
LET pm_q_prop_inv =
λ (s_1: State): λ (res: StmtResult[State]): pm_q_prop(pm)(catch_break(res))
IN
pm`states(s) ∧
R(s)(skip(s)) ∧
while_variant?(b_ex, body, <)(variant(s), R(s)) ∧
while_invariant?(b_ex, body)(pm_q_prop_inv(s), pm_q_prop_inv(s))
⊃ pm_q_prop(pm)(while(b_ex, body)(P, Q, R, variant, <)(s))

Repeatedly Skolemizing and flattening,
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma hoare_while[State, Ord],
Rewriting using while_to_while_no_cb, matching in *,
Expanding the definition of ##,

Expanding the definition of while_no_cb,

Expanding the definition of skip,

Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,

Expanding the definition of pm_q_prop,

Expanding the definition of catch_break,

which is trivially true.

This completes the proof of `pm_q_prop_while`.

Q.E.D.

C.241.12 While_Rewrites.pm_q_prop_stmt_while_TCC1

Terse proof for `pm_q_prop_stmt_while_TCC1`.

`pm_q_prop_stmt_while_TCC1:`

```
{1}  ∀ (pm: Plain_Memory[State], s: State, stmt: [State → StmtResult[State]],
          pm_q_prop_inv: [State → [StmtResult[State] → bool]]):
        (pm_q_prop(pm)(stmt(s)) ∧
          pm_q_prop_inv =
           (λ (s_1: State):
                λ (res: StmtResult[State]): pm_q_prop(pm)(catch_break(res))))
         ⊃
         OK?[State](stmt(s)) ∨
          Break?[State](stmt(s)) ∨
           Continue?[State](stmt(s)) ∨
            Return?[State](stmt(s)) ∨
             Switch?[State](stmt(s)) ∨
              Default?[State](stmt(s)) ∨ Exception?[State](stmt(s))
```

Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of `pm_q_prop_stmt_while_TCC1`.

Q.E.D.

C.241.13 While_Rewrites.pm_q_prop_stmt_while

Terse proof for `pm_q_prop_stmt_while`.

`pm_q_prop_stmt_while`:

{1} ∀ (<: PRED[[Ord, Ord]], P, Q, R: [State → PRED[StmtResult[State]]], b_ex: [State → ExprResult[State, bool]], body: [State → StmtResult[State]], pm: Plain_Memory[State], s: State, stmt: [State → StmtResult[State]], variant: [State → [State → Ord]]):
LET pm_q_prop_inv = λ (s_1: State): λ (res: StmtResult[State]): pm_q_prop(pm)(catch_break(res))
IN
pm_q_prop(pm)(stmt(s)) ∧
R(state(stmt(s)))(stmt(s)) ∧
while_variant?(b_ex, body, <)(variant(state(stmt(s))), R(state(stmt(s)))) ∧
while_invariant?(b_ex, body)
(pm_q_prop_inv(state(stmt(s))),
pm_q_prop_inv(state(stmt(s))))
⊃ pm_q_prop(pm)((stmt ## while(b_ex, body)(P, Q, R, variant, <))(s))

Repeatedly Skolemizing and flattening,
Installing automatic rewrites from: ## skip lift!
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Using lemma pm_q_prop_while,
Applying eta axiom scheme to stmt!1(s!1),
Repeatedly simplifying with BDDs, decision procedures, rewriting, and if-lifting,
Expanding the definition of pm_q_prop,
which is trivially true.
This completes the proof of `pm_q_prop_stmt_while`.
Q.E.D.