Coalgebraic Methods for Object–Oriented Specification

Hendrik Tews

Coalgebraische Methoden für Objektorientierte Spezifikation

Dissertation

zur Erlangung des akademischen Grades
Doktor rerum naturalium (Dr. rer. nat.)

vorgelegt an der
Technischen Universität Dresden, Fakultät Informatik

eingereicht von

Diplom–Informatiker Hendrik Tews
geboren am 22. August 1969 in Leipzig

Gutachter : Prof. Dr. rer. nat. habil. Horst Reichel (TU Dresden)
Prof. Dr. Bart Jacobs (Katholieke Universiteit Nijmegen)
Prof. Dr. rer. nat. habil. Heinrich Hußmann (TU Dresden)

verteidigt am 18. Oktober 2002 in Dresden

Dresden 5. Oktober 2002

Acknowledgements

First I would like to thank Prof. Horst Reichel. I have been working under his supervision since my fourth year's project. In all that time I benefited from many fruitful discussions with Horst Reichel. Without his continuing support this thesis would not exist.

I am indebted to Bart Jacobs, the driving force behind the LOOP project. He accompanied all my work on this thesis through its different stages. In numerous discussions Bart clarified my vague ideas and taught me (fibred) category theory. There are only a few pages in this thesis that have not been shaped by his influence.

I am grateful to Ulrich Hensel. It was a lot of fun exploring the world of categories and coalgebras together with Ulrich. He suggested the idea of the CCSL compiler to me and contributed much to its early versions.

Some parts of the research described in this thesis were performed during my visit to the University of Nijmegen, the Netherlands. I want to thank the group of Frits Vaandrager at this university for their support during my stay in Nijmegen.

Many people gave generously of their time and intellect, reading and patiently correcting various drafts of this PhD thesis. I thank Frank Felfe, Michael Hohmuth, Bart Jacobs, Alexander Kurz, Jan Rothe, and Marc Schroeder for their detailed comments. I also thank the three reviewers for their suggestions.

I am indebted to the developers of the theorem prover PVS. Without their excellent tool, some of the results and many of the examples described here would still be unknown.

I thank Brendan McIntyre for doing the cover design.

Contents

List of Figures and Tables

Deutsche Kurzfassung (Extended Abstract in German)

Die vorliegende Dissertation beschreibt *coalgebraische* Mittel und Methoden zur Softwarespezifikation und –verifikation. Die Ergebnisse dieser Dissertation vereinfachen die Anwendung coalgebraischer Spezifikations– und Verifikationstechniken und erweitern deren Anwendbarkeit. Damit werden Softwareverifikation im Allgemeinen und im Besonderen coalgebraische Methoden zur Softwareverifikation der praktischen Anwendbarkeit ein Stück nähergebracht.

Diese Dissertation enthält zwei wesentliche Beiträge:

- Im Kapitel 3 wird eine Erweiterung des klassischen Begriffs der Coalgebra vorgestellt. Diese Erweiterung erlaubt die coalgebraische Modellierung von Klassenschnittstellen mit beliebigen Methodentypen (insbesondere mit binären Methoden).

- Im Kapitel 4 wird die coalgebraische Spezifikationssprache CCSL (*Coalgebraic Class Specification Language*) vorgestellt. Die Bescheibung umfasst Syntax, Semantik und einen Prototypcompiler, der CCSL Spezifikationen in Logik höherer Ordnung (passend für die Theorembeweiser PVS und ISABELLE/HOL) übersetzt.

Coalgebraische Methoden, so wie sie in dieser Dissertation beschrieben werden, haben —oberflächlich betrachtet— eine gewisse Ähnlichkeit mit algebraischen Spezifikations– und Verifikationsmethoden. In beiden Ansätzen werden die operationalen Schnittstellen mit Hilfe von Signaturen beschrieben. Mit Hilfe einer geeigneten Logik werden die semantischen Eigenschaften der Schnittstelle erfasst.

Das Ziel der Forschung zu Coalgebren besteht nicht darin, algebraische Methoden durch coalgebraische zu ersetzen, sondern vielmehr in der Integration beider Ansätze in einer einheitlichen Umgebung. Eine solche Umgebung erlaubt die Vorteile sowohl des algebraischen als auch des coalgebraischen Ansatzes zur Softwarespezifikation auszunutzen. Von einem Softwaresystem könnten dann zum Beispiel die dynamischen Aspekte mit coalgebraischen Methoden beschrieben werden, während die (statischen) Daten, die im System ausgetauscht werden, mit algebraischen Techniken modelliert werden.

Die Spezifikationssprache CCSL (zusammen mit einem der Theorembeweiser PVS oder ISABELLE/HOL) bietet eine solche Integration algebraischer und coalgebraischer Werkzeuge. In CCSL können einerseits abstrakte Datentypen mit initialer Semantik definiert werden. Andererseits stellt CCSL Klassenspezifikationen mit coalgebraischen Signaturen und einer speziellen coalgebraischen Logik bereit.

Algebren und Coalgebren: Eine kurze Einführung

Bevor ich mit der Beschreibung der konkreten Ergebnisse der vorliegenden Dissertation fortfahre, möchte ich zunächst die wesentlichen Schlagwörter erläutern. Eine *Algebra* ist eine Funktion mit einem strukturierten Definitionsbereich, wie zum Beispiel

$$\mathsf{cons} : \ A \times \mathsf{List}[A] \longrightarrow \mathsf{List}[A]$$

Eine *algebraische Signatur* besteht aus einer endlichen Menge von Algebradeklarationen. Eine *algebraische Spezifikation* bereichert eine algebraische Signatur um eine Menge von Axiomen, die zusätzliche Eigenschaften der algebraischen Signatur formalisieren. Ein *Modell* einer algebraischen Spezifikation besteht aus einer Menge von Funktionen —den Interpretationen der algebraischen Signatur— die die Axiome erfüllen. Das Forschungsgebiet Algebraische Spezifikation stellt sich zum Ziel (Teile von) Software mittels Algebren zu beschreiben und deren Eigenschaften mit einer geeigneten Logik zu beweisen. Endlich generierte Datenstrukturen, wie Listen oder binäre Bäume sowie Funktionen, die solche Datenstrukturen verarbeiten, können mit Hilfe von algebraischer Spezifikation sehr elegant beschrieben werden. Die Modellierung findet dabei auf abstraktem Niveau, ohne Bezug auf eine konkrete Implementierung, statt.

Unter Verifikation versteht man die Entwicklung von mathematischen Beweisen, die zeigen, dass sich die modellierte Software so verhält, wie spezifiziert. Eine Verifikation kann verschiedene Dinge umfassen, wie zum Beispiel das Ableiten von Eigenschaften aus den Axiomen einer Spezifikation oder die Konstruktion von Modellen sowie die Konstruktion von Verfeinerungen einzelner Modelle. Die einzelnen Schritte einer Verifikation sind vom mathematischen Standpunkt aus gesehen oft sehr einfach. Eine Verifikation im Ganzen ist hingegen oft unerwartet komplex, auf Grund der großen Menge von Einzelschritten und Fallunterscheidungen. Realistische Verifikationsbeispiele erfordern deshalb die Benutzung von *Theorembeweisern.* Ein Theorembeweiser ist ein Softwarewerkzeug, das eine Logik zusammen mit einem Ableitungssystem implementiert. Der Theorembeweiser kann zum Finden und Überprüfen von Beweisen eingesetzt werden. Während der Verifikation kann dem Theorembeweiser der stupide Teil der Arbeit übertragen werden (zum Beispiel das Überprüfen von Fallunterscheidungen auf Vollständigkeit oder das Ausführen von Berechnungen). Die Person, die die Verifikation ausführt, kann sich dann auf die interessanten und schwierigen Aspekte der Verifikation konzentrieren. In der vorliegenden Dissertation betrachte ich die Theorembeweiser PVS (Owre et al., 1996) und ISABELLE/HOL (Nipkow et al., 2002b; Nipkow et al., 2002a). Beide Systeme sind interaktive Theorembeweiser für Logik höherer Stufe.

Eine *Coalgebra* ist eine Funktion mit strukturiertem Wertebereich, wie zum Beispiel[1]

$$\mathsf{step} : \ \mathsf{Aut}[A, B] \longrightarrow A \Rightarrow (\mathsf{Aut}[A, B] \times B)$$

[1] Im Folgenden bezeichnet $A \Rightarrow (\mathsf{Aut}[A, B] \times B)$ die Menge der Funktionen $A \longrightarrow \mathsf{Aut}[A, B] \times B$, die häufig auch als $(\mathsf{Aut}[A, B] \times B)^A$ dargestellt wird.

oder dazu äquivalent[2]

$$\mathsf{step}' : \ \mathsf{Aut}[A,B] \times A \longrightarrow \mathsf{Aut}[A,B] \times B$$

Betrachten Sie jetzt $\mathsf{Aut}[A,B]$ als die Menge der (nicht notwendigerweiser endlichen) Automaten mit Eingabealphabet A und Ausgabealphabet B. Dann ist step (beziehungsweise step') die Zustandsüberführungsfunktion, die einen Zustand eines Automaten zusammen mit einer Eingabe auf einen Nachfolgezustand und eine Ausgabe abbildet. Im Zusammenhang mit Coalgebren wird die Menge $\mathsf{Aut}[A,B]$ als der *Zustandsraum* der Coalgebra step bezeichnet. Die Elemente des Zustandsraumes werden als *black box* betrachtet, deren interner Zustand nicht sichtbar ist. Die Coalgebra step bietet somit die einzige Möglichkeit, einzelne Zustände zu untersuchen. Die einmalige Anwendung der Coalgebra für eine spezifische Eingabe aus A liefert eine sichtbare Ausgabe in B und einen Nachfolgezustand mit dem das Experiment bis ins Unendliche forgesetzt werden kann.

Die Menge aller sichtbaren Ausgaben, die man für einen Zustand $x \in \mathsf{Aut}[A,B]$ durch beliebig langes Anwenden der Coalgebra mit allen möglichen Eingaben erhält, ist das *sichtbare Verhalten* von x. Es ist insbesondere möglich, dass zwei verschiedene Zustände das gleiche sichtbare Verhalten zeigen. In diesem Fall sind die zwei Zustände *verhaltensäquivalent*. Der Begriff der *Bisimulation* formalisiert das intuitive Konzept der Verhaltensgleichheit. Eine Bisimulation ist eine Relation auf dem Zustandsraum einer Coalgebra, die Zustände mit gleichem sichtbaren Verhalten in Beziehung setzt. Desweiteren ist eine Bisimulation gegenüber der Bildung von Nachfolgezuständen abgeschlossen. Im Anwendungsbereich von Coalgebren spielen Bisimulationen eine große Rolle, da man im Wesentlichen an der Verhaltensgleichheit von Zuständen interessiert ist. Ob zwei gegebene Zustände gleich sind, ist hingegen für gewöhnlich nicht von Interesse. Für Coalgebren wird in der praktischen Arbeit deshalb Gleichheit häufig durch Bisimulation (oder Bisimilarität) ersetzt.

Coalgebren können auf sehr einfache Weise Ausnahmebedingungen (*exceptions*) oder partielles Verhalten modellieren. Betrachten Sie zum Beispiel die folgende Coalgebra:[3]

$$\mathsf{step}_p : \ \mathsf{PAut}[A,B] \longrightarrow A \Rightarrow (\mathsf{PAut}[A,B] \times B \ + \ \mathbf{1})$$

oder, äquivalenterweise,

$$\mathsf{step}_p' : \ \mathsf{PAut}[A,B] \times A \longrightarrow \mathsf{PAut}[A,B] \times B \ + \ \mathbf{1}$$

Elemente von $\mathsf{PAut}[A,B]$ sind (potentiell unendliche) partielle Automaten. Abhängig vom Zustand kann für bestimmte Eingaben aus A die Zustandsüberführungsfunktion

[2] Die Menge der Funktionen $X \longrightarrow Y \Rightarrow Z$ korrespondiert eineindeutig mit der Menge $X \times Y \longrightarrow Z$.

[3] Die Operation $+$ steht hier für die disjunkte Vereinigung und $\mathbf{1} = \{*\}$ bezeichnet die einelementige Menge.

undefiniert sein. In diesem Fall liefert step_p den Wert $* \in \mathbf{1}$ als Ergebnis. Dem allgemeinen Sprachgebrauch folgend würde man sagen, der Automat stürzt ab.

Eine *Coalgebraische (Klassen–) Spezifikation* besteht aus einer endlichen Menge von Coalgebra Deklarationen —der *coalgebraischen Signatur*— und einer endlichen Menge von Axiomen. Auf die Logik, die zur Formulierung der Axiome genutzt wird, gehe ich im Folgenden noch genauer ein (die präzise Beschreibung ist in Kapitel 4). Axiome in coalgebraischen Spezifikationen schränken typischerweise das Verhalten der Coalgebren ein. Zum Beispiel:[4]

$$\forall x \in \mathsf{PAut}[A, B] \ . \quad \mathsf{step}_p\, x\, a_0 \ \neq \ *$$

Dieses Axiom beschreibt, dass es keinen partiellen Automaten gibt, der bei Eingabe von a_0 abstürzt. Axiome fordern oft auch die Verhaltensäquivalenz bestimmter Zustände. Das folgende Axiom beschreibt, dass das Verhalten aller Automaten zyklisch ist.

$$\forall x \in \mathsf{Aut}[A, B] \ . \ \forall a_0, a_1 \in A \ . \quad x \ \underline{\leftrightarrow} \ \pi_1\big(\mathsf{step}\ (\pi_1\ (\mathsf{step}\, x\, a_0))\ \ a_1\big)$$

(Hier bezeichnet $\pi_1 : \mathsf{Aut}[A, B] \times B \longrightarrow \mathsf{Aut}[A, B]$ die erste Projektion und das Symbol $\underline{\leftrightarrow}$ steht für die größte Bisimulation.)

Neben Bisimulationen spielen *Coalgebramorphismen* und *Invarianten* eine wichtige Rolle für Coalgebren. Ein Coalgebramorphismus ist eine strukturerhaltende Abbildung zwischen Coalgebren. Eine Invariante ist eine Eigenschaft (eines Zustandes einer Coalgebra), die, falls sie einmal gilt, für alle direkt und indirekt erreichbaren Nachfolgezustände gültig bleibt. Invarianten spielen eine wichtige Rolle bei der Definition von coalgebraischen Logiken und bei der Konstruktion von Verfeinerungen für coalgebraische Spezifikationen (eine ausführliche Darstellung coalgebrischer Verfeinerung ist in (Jacobs and Tews, 2001)).

Die meisten Autoren definieren sowohl den Begriff der Bisimulation als auch den der Invariante mit Hilfe von Coalgebramorphismen. Die Definition von Coalgebramorphismen beruht auf der Struktur der Signatur.

Mittel und Methoden

Die vorliegende Dissertation untersucht Fragestellungen, die sich aus dem Ansatz ergeben, Software mit Hilfe von Algebren *und* Coalgebren zu modellieren. Im Konkreten liegt Software als ein Algorithmus vor, der in einer bestimmten Programmiersprache verfasst wurde. Ein allgemeines Problem hierbei ist, dass sich Programmiersprachen oft sehr stark von der in der Mathematik verwendeten Mengenlehre unterscheiden. Die Unterschiede betreffen sowohl die Konstruktionen, die in einer Programmiersprache beziehungsweise in Mengenlehre möglich sind, als auch die Eigenschaften, die gelten. Man kann zum Beispiel für zwei Mengen A und B immer die Menge der Abbildungen $A \Rightarrow B$ bilden.

[4] Wie in HOL und in funkionalen Sprachen üblich, schreibe ich Funktionsapplikation *ohne* explizite Klammern. Funktionsapplikation ist linksassoziativ, das heißt, es gilt $\mathsf{step}_p\, x\, a_0 = (\mathsf{step}_p(x))(a_0)$.

Die Bildung von Funktionstypen ist jedoch nur in bestimmten Programmiersprachen möglich. Ein weiteres allgemeines Problem ist, dass Programmiersprachen untereinander oft sehr unterschiedlich sind.

Aus den genannten Gründen ist es sinnvoll, nicht mit Mengenlehre zu arbeiten, sondern mit einem abstrakterem Formalismus, der eine bessere Kontrolle über die zur Verfügung stehenden Konstruktionen und die geltenden Eigenschaften erlaubt. Einen solchen Formalismus bietet die *Kategorientheorie* (Mac Lane, 1997). Eine Kategorie kann man als Abstraktion eines Universums von Mengen auffassen. Eine Kategorie enthält Objekte (als Abstraktion von Mengen) und Morphismen (als Abstraktion von Funktionen). Im Allgemeinen enthält eine Kategorie nur wenig Struktur und besitzt nur einfachste Eigenschaften. Um zum Beispiel mit geordneten Paaren arbeiten zu können, ist es notwendig, die Existenz von *kategorientheoretischen Produkten* vorauszusetzen. Auf diese Weise ist es möglich, eine Kategorie sehr genau (in Bezug auf Eigenschaften und verfügbare Konstruktionen) an eine gegebene Programmiersprache anzupassen.

Auf dem Abstraktionsniveau von Kategorientheorie übernehmen *Endofunktoren* die Rolle von Signaturen. (Ein Endofunktor ist eine strukturerhaltende Abbildung einer Kategorie auf sich selbst.) Eine Coalgebra in einer Kategorie $\mathbb{C}$ ist dann ein Morphismus $X \longrightarrow F(X)$ in $\mathbb{C}$, wobei X —der Zustandsraum der Coalgebra— ein Objekt aus $\mathbb{C}$ ist. Der Endofunktor $F : \mathbb{C} \longrightarrow \mathbb{C}$ formalisiert dabei die coalgebraische Signatur. Eine Algebra ist, analogerweise, ein Morphismus der Form $F(X) \longrightarrow X$. Die einfache Typstruktur gängiger Programmiersprachen ermöglicht es, sich bei der Modellierung von Software auf Signaturen mit einfacher Struktur einzuschränken. Als weitere Konsequenz ist es daher ausreichend, nur eine stark eingeschränkte Klasse von Funktoren zu betrachten. Für viele Anwendungen sind *polynomiale Funktoren* ausreichend. Die Klasse der polynomialen Funktoren wird durch die folgende Grammatik generiert (siehe auch Abschnitt 2.6.1).

$$F(X) \;=\; A \;\mid\; X \;\mid\; F_1(X) \times F_2(X) \;\mid\; F_1(X) + F_2(X) \;\mid\; A \Rightarrow F_1(X)$$

Hierbei steht A für eine beliebige Menge, $\times$ bezeichnet das (kartesische) Produkt, $+$ das Coprodukt (die disjunkte Vereinigung) und $\Rightarrow$ den Exponenten (Funktionsraum).

Für die Logik, die zur Verifikation der Software benutzt wird, gilt ein ähnliches Argument in Bezug auf das Abstraktionsniveau. Die verschiedenen Theorembeweiser stellen unterschiedliche Logiken mit unterschiedlichen Eigenschaften zur Verfügung. Es ist deshalb sinnvoll, mit einer Abstraktion mathematischer Logik zu arbeiten. Die Theorie der *Fibrationen* (Jacobs, 1999a; Phoa, 1992) stellt eine solche Abstraktion innerhalb von Kategorientheorie zur Verfügung.

Aus den eben dargestellten Gründen beruhen die zentralen Kapitel der vorliegenden Dissertation in nichttrivialer Weise auf der Kategorientheorie. Das Einführungskapitel, Kapitel 2, enthält alle Definitionen und erklärt den über einfache Mengenlehre hinausgehenden Teil der in dieser Dissertation verwendeten Notation. Im Einführungskapitel werden in Abschnitt 2.4 zwei konkrete Fibrationen im Detail vorgestellt: Die Fibration der (getypten) Prädikate sowie die der (getypten) Relationen. Beide Fibrationen sind essentiell für Kapitel 3. Die Abschnitte 2.5 und 2.6 enthalten eine kurze Einführung in Algebren und Coalgebren. Der Abschnitt 2.6 über Coalgebren enthält eine Sammlung bekannter Resultate für Coalgebren polynomialer Funktoren. Diese Resultatssammlung spielt eine wichtige Rolle für Kapitel 3, in dem ich verschiedene Verallgemeinerungen polynomialer Funktoren diskutiere.

Ergebnisse

Im Folgenden gehe ich auf einige Aspekte dieser Dissertation näher ein. Der erste Unterabschnitt behandelt die Ergebnisse die ich bei der Generalisierung polynomialer Funktoren erzielen konnte. Der folgende Unterabschnitt stellt die coalgebraische Spezifikationssprache CCSL vor.

Binäre Methoden

Im Kapitel 3 beschäftige ich mich mit dem Problem wie Methoden beliebigen Types, insbesondere binäre Methoden, mit Hilfe von Coalgebren modelliert werden können. Die Begriffe *Methode* und *binäre Methode* stammen aus der objektorientierten Programmierung. Im objektorientierten Ansatz wird Software mit Hilfe von Klassen modularisiert. Eine Klasse repräsentiert eine Abstraktion bestimmter Daten zusammen mit typischen Operationen. Diese Operationen werden Methoden genannt. Jede Methode gehört zu einer Klasse. Die Laufzeitdatenstrukturen von Klassen sind *Objekte*. Berechnungen bestehen im Wesentlichen aus Methodenaufrufen, die typischerweise als $o.f$ geschrieben werden. Dabei ist o ein Objekt und f eine Methode aus der Klasse zu der o gehört. Beim Aufruf der Methode f bekommt f ein zusätzliches (oft implizites) Argument, das Objekt o. Eine Methode ist eine binäre Methode, wenn sie noch mindestens ein weiteres Objekt der gleichen Klasse als Argument erhält. Auf die Datenfelder eines Objektes hat man üblicherweise von außen keinen Zugriff. Es gibt nur die Möglichkeit Methoden aufzurufen. Damit können Implementierungsdetails versteckt werden. Aus diesem Grund eignen sich Klassen gut zur Datenabstraktion.

Der Aspekt der Datenabstraktion und das Berechnungsmuster legen es nahe Coalgebren zur Modellierung von Klassen in objektorientierten Programmiersprachen zu verwenden. Diese Idee geht auf (Reichel, 1995) zurück. Allerdings unterlag dieser Ansatz bis jetzt einer erheblichen Beschränkung: Klassen, die binäre Methoden enthalten, können nicht als Coalgebren (für Endofunktoren) modelliert werden.

Lassen Sie mich als Beispiel die Schnittstelle der zuvor eingeführten Automaten um die Operation merge erweitern. Die Funktion merge soll aus zwei Automaten a und b das Interleaving bilden: Der aus der Anwendung $\mathsf{merge}(a, b)$ resultierende Automat benutzt abwechselnd a und b um die Ausgaben zu produzieren. Der Typ von merge ist offensichtlich:

$$\mathsf{merge} : \ \mathsf{Aut}[A, B] \times \mathsf{Aut}[A, B] \longrightarrow \mathsf{Aut}[A, B]$$

Unter Ausnutzung der Tatsache, dass jede Funktion $X \times Y \longrightarrow Z$ auf eineindeutige Weise einer Funktion $X \longrightarrow Y \Rightarrow Z$ entspricht, kann der Typ von merge in coalgebraische Form gebracht werden.

$$\mathsf{merge} : \; \mathsf{Aut}[A,B] \xrightarrow{\qquad} \mathsf{Aut}[A,B] \Rightarrow \mathsf{Aut}[A,B]$$

Hier sieht man deutlich, dass der Zustandsraum $\mathsf{Aut}[A,B]$ in einer contravarianten Position (auf der linken Seite von $\Rightarrow$) im Wertebereich von merge auftritt. Aus diesem Grund kann die Signatur der Automaten zusammen mit der merge Operation nicht mit einem polynomialen Funktor modelliert werden.[5]

In Kapitel 3 stelle ich eine Lösung für das eben beschriebene Problem mit binären Methoden vor. Die Lösung basiert darauf, polynomiale Funktoren zu bivarianten Funktoren $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}$ zu generalisieren.[6] Die allgemeinste Klasse solcher Funktoren, die in dieser Arbeit untersucht wird, ist die Klasse der polynomialen Funktoren höherer Ordnung (siehe Abschnitt 3.2). Polynomiale Funktoren höherer Ordnung werden durch die folgende Grammatik generiert:

$$\begin{aligned} H(Y,X) \;=\; & A \;\mid\; X \;\mid\; H_1(Y,X) \times H_2(Y,X) \;\mid \\ & \qquad\qquad H_1(Y,X) + H_2(Y,X) \;\mid\; H_1(X,Y) \Rightarrow H_2(Y,X) \end{aligned}$$

Durch den allgemeinen Exponenten können polynomiale Funktoren höherer Ordnung Signaturen mit beliebigen Methodentypen, insbesondere auch mit binären Methoden, modellieren. Im Folgenden werde ich den Begriff der Coalgebra entsprechend generalisieren und einige Ergebnisse aus Kapitel 3 vorstellen.

Definition 1 *Sei $H : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ ein polynomialer Funktor höherer Ordnung. Eine H–Coalgebra ist dann eine Funktion $X \longrightarrow H(X,X)$. Ein H–Coalgebramorphismus zwischen den Coalgebren $c : X \longrightarrow H(X,X)$ und $d : Y \longrightarrow H(Y,Y)$ ist eine Funktion $f : X \longrightarrow Y$ für die das folgende Diagramm kommutiert:*

$$\begin{tikzcd}
X \arrow[r, "c"] \arrow[dd, "f"'] & H(X,X) \arrow[dr, "H(X,f)"] & \\
& & H(X,Y) \\
Y \arrow[r, "d"] & H(Y,Y) \arrow[ur, "H(f,Y)"'] &
\end{tikzcd}$$

Diese Definition ist eine konservative Erweiterung des bekannten Begriffes der Coalgebra: Betrachtet man einen polynomialen Funktor F als Funktor höherer Ordnung,

[5] Genauer gesagt, kann die coalgebraische Signatur von merge nur für die Kategorien mit einem Endofunktor modelliert werden, in denen jeder Morphismus umkehrbar ist.

[6] In dieser Kurzfassung beschränke ich mich auf die Kategorie der Mengen und totalen Funktionen.

dann stimmt die Definition von F–Coalgebren nach Definition 1 mit dem bekannten Begriff der Coalgebra überein und das obige Fünfeck schrumpft zum bekannten Quadrat.

Die generalisierten Definitionen für Bisimulationen und Invarianten befinden sich im Abschnitt 3.3 (Definition 3.3.3 auf Seite 85). Bei der Definition von Bisimulationen und Invarianten folge ich dem Ansatz von (Hermida and Jacobs, 1998): Zuerst verallgemeinere ich in Definition 3.3.1 (auf Seite 83) das Liften von Predikaten und Relationen für polynomiale Funktoren höherer Stufe. Mit Hilfe dieser Liftings werden dann Bisimulationen und Invarianten definiert.

Der weit verbreitete und auf (Aczel and Mendler, 1989) zurückgehende Ansatz, Bisimulationen mit Hilfe von Coalgebramorphismen zu definieren, führt für polynomiale Funktoren höherer Ordnung nicht zum Ziel. Der daraus resultierende Begriff der Bisimulation wäre gegenüber dem Bilden von Nachfolgezuständen nicht abgeschlossen.

Es zeigt sich, dass mit der Generalisierung zu polynomialen Funktoren höherer Ordnung fast alle bekannten Eigenschaften von Coalgebren verlorengehen. Abschnitt 3.3 enthält nur drei positive Resultate (siehe auch Proposition 3.3.6).

Satz 2 *Für eine Coalgebra eines polynomialen Funktors höherer Ordnung mit Zustandsraum X ist die Menge X eine Invariante und die Gleichheitsrelations auf X eine Bisimulation. Desweiteren gilt: Falls R eine Bisimulation ist, dann ist auch $R^{\mathrm{op}} = \{(y, x) \mid x\, R\, y\}$ eine Bisimulation.*

Alle weiteren für Coalgebren polynomialer Funktoren bekannten Resultate gelten im Allgemeinen nicht.

Beobachtung 3 *Für Coalgebren polynomialer Funktoren höherer Ordnung gilt* keiner *der folgenden Punkte.*

1. *Bisimulationen und Invarianten sind unter mengentheoretischer Vereinigung abgeschlossen.*
2. *Bisimulationen und Invarianten sind unter mengentheoretischem Durchschnitt abgeschlossen.*
3. *Die relationale Komposition von zwei Bisimulationen ist eine Bisimulation.*
4. *Der Graph eines Coalgebramorphismus ist eine Bisimulation.*
5. *Für einen Coalgebramorphismus $f : X \longrightarrow Y$ ist das direkte Bild $\coprod_f \top = \{f\, x \mid x \in X\}$ eine Invariante.*
6. *Invarianten definieren Teilcoalgebren und umgekehrt.*
7. *Die Relation $\coprod_\delta P = \{(x, x) \mid x \in P\}$ ist eine Bisimulation falls P eine Invariante ist.*

8. *Das Prädikat $\coprod_{\pi_1} R = \{x \mid \exists y \,.\, x\, R\, y\}$ ist eine Invariante falls R eine Bisimulation ist.*

9. *Die Relation ${\pi_1}^* P \;\wedge\; R = \{(x, y) \mid x\, R\, y \wedge x \in P\}$ ist eine Bisimulation für alle Invarianten P und Bisimulationen R.*

10. *Der Kern einen Coalgebramorphismus ist eine Bisimulation.*

Die Gegenbeispiele, die Beobachtung 3 belegen, sind überraschend einfach strukturiert. In allen Fällen enthält der Zustandsraum der Coalgebren weniger als sechs Elemente. Ein Teil der Gegenbeispiele ist in Kapitel 3 zu finden. Die Restlichen sind im PVS Material zu dieser Dissertation enthalten, siehe Anhang A.

Mit Ausnahme von Punkt 1 basieren alle Gegenbeispiele auf dem Funktor $T(Y, X) = (X \Rightarrow Y) \Rightarrow X$ (siehe Beispiel 3.3.8). Diese Beobachtung führt zur Definition von weiteren Klassen von Funktoren.

Kartesische Funktoren:

$$K(X) \;=\; A \;\mid\; X \;\mid\; K_1(X) \times K_2(X) \;\mid\; K_1(X) + K_2(X)$$

Erweitert Kartesische Funktoren:

$$\begin{aligned} G^K(Y, X) \;=\; & A \;\mid\; X \;\mid\; G_1^K(Y, X) \times G_2^K(Y, X) \;\mid \\ & \qquad G_1^K(Y, X) + G_2^K(Y, X) \;\mid\; K(Y) \Rightarrow G_1^K(Y, X) \end{aligned}$$

Erweitert Polynomiale Funktoren:

$$\begin{aligned} G^F(Y, X) \;=\; & A \;\mid\; X \;\mid\; G_1^F(Y, X) \times G_2^F(Y, X) \;\mid \\ & \qquad G_1^F(Y, X) + G_2^F(Y, X) \;\mid\; F(Y) \Rightarrow G_1^F(Y, X) \end{aligned}$$

Alle hier vorgestellten Funktorklassen unterscheiden sich jeweils nur in der Klausel für den Exponenten. Die Klasse der kartesischen Funktoren dient nur der Definition von erweitert kartesischen Funktoren. Erweitert kartesische Funktoren sind die polynomialen Funktoren höherer Ordnung, bei denen der Definitionsbereich in der Klausel für den Exponenten mit einem kartesischen Funktor dargestellt werden kann. Bei erweitert polynomiale Funktoren ist ein polynomialer Funktor in Definitionsbereich zulässig. Tabelle 1 zeigt die verschiedenen Funktorklassen mit Beispielen zur Ausdrucksstärke.

Die Beschränkung des Exponenten führt zu stärkeren strukturellen Eigenschaften.

Satz 4 *Für Coalgebren erweitert polynomialer Funktoren gelten bis auf Punkt 1 alle in Beobachtung 3 genannten Eigenschaften.*

Funktorklasse	Eigenschaft	Beispiel
kartesisch	keine Argumente	$\mathsf{Self} \longrightarrow \boxed{\cdots}$
polynomial	konstante Argumente	$\mathsf{Self} \times A \longrightarrow \boxed{\cdots}$
erweitert kartesisch	keine funktionalen Argumente	$\mathsf{Self} \times \mathsf{Self} \times \mathsf{Self} \longrightarrow \boxed{\cdots}$ $\mathsf{Self} \times (\mathsf{Self} + A) \longrightarrow \boxed{\cdots}$
erweitert polynomial	funktionale Argumente mit konst. Def.-bereich	$\mathsf{Self} \times (A \Rightarrow \mathsf{Self}) \longrightarrow \boxed{\cdots}$
polynomial höherer Stufe	beliebige Argumente	$\mathsf{Self} \times (\mathsf{Self} \Rightarrow A) \longrightarrow \boxed{\cdots}$ $\mathsf{Self} \times (\mathsf{Self} \Rightarrow \mathsf{Self}) \longrightarrow \boxed{\cdots}$

Tabelle 1.: Klassen von Funktoren im Kapitel 3

Für erweitert polynomiale Funktoren lässt sich außerdem zeigen, dass das Lifting von Prädikaten und das von Relationen gefasert ist, siehe Lemma 3.4.4 und 3.4.7. Desweiteren führen die beiden verschiedenen Ansätze zur Definition von Bisimulationen und Invarianten nach (Hermida and Jacobs, 1998) und nach (Aczel and Mendler, 1989) zu gleichen Resultaten (siehe Abschnitt 3.4.5).

Für Coalgebren erweitert polynomialer Funktoren verbleibt das Manko, dass sowohl Bisimulationen als auch Invarianten bezüglich ihrer Vereinigung nicht abgeschlossen sind. Die Situation lässt sich mit enger gefassten Definition zum Teil verbessern: In Abschnitt 3.4.6 werden *starke Invarianten* definiert. Starke Invarianten bilden einen vollständigen Verband bezüglich mengentheoretischer Vereinigung und mengentheoretischem Durchschnitt. In Abschnitt 3.4.7 zeige ich, dass die reflexiven Bisimulationen einen (unvollständigen) Verband bilden. Das heißt, zu jeweils zwei reflexiven Bisimulation gibt es eine (reflexive) Bisimulation, die größer ist als deren Vereinigung. Eine größte Bisimulation existiert allerdings im Allgemeinen nicht, siehe Beispiel 3.5.10.

Mit der weiteren Einschränkung auf erweitert kartesische Funktoren lässt sich ein Resultat von (Poll and Zwanenburg, 2001) übertragen und leicht generalisieren (siehe auch Abschnitt 3.5).

Satz 5 *Für Coalgebren erweitert kartesischer Funktoren bilden die Bisimulationsäquivalenzen (das heißt, die Bisimulationen, die Äquivalenzrelationen sind) einen vollständigen Verband. Insbesondere existiert eine größte Bisimulationsäquivalenz.*

Auf die Existenz finaler Coalgebren wird im Abschnitt 3.6 eingegangen. Sobald die Signatur eine binäre Methode enthält, das heißt, sobald die Signatur nicht mehr einem polynomialen Funktor entspricht, lässt sich mit Hilfe eines Diagonalisierungsargumentes zeigen, dass eine finale Coalgebra *nicht* existieren kann. Allerdings kann es in eingeschränkten Klassen von Coalgebren (die zum Beispiel der Semantik einer coalgebraischen Spezifikation entsprechen) durchaus finale Coalgebren geben.

Zusammenfassend lässt sich sagen, dass Coalgebren erweitert polynomialer Funktoren genügend strukturelle Eigenschaften besitzen, um als Semantik für Klassen objektorientierter Sprachen zu dienen. Für die Modellierung vieler objektorientierter Sprachen sind sogar erweitert kartesische Funktoren ausreichend.

Die Ergebnisse von Kapitel 3 wurden zum Teil schon in (Tews, 2000b; Tews, 2001) sowie in (Tews, 2002b) veröffentlicht.

Die Coalgebraische Spezifikationssprache CCSL[7]

Sowohl Algebren als auch Coalgebren haben, wenn sie zur Modellierung von Software verwendet werden, bestimmte Vor– und Nachteile. Coalgebren eignen sich gut zur Darstellung möglicherweise nicht terminierender Systeme. Mit Algebren lassen sich endlich erzeugte Datentypen wie Listen oder Bäume elegant modellieren. Softwaresysteme beinhalten im Allgemeinen sowohl Datentypen als auch Prozesse. Für eine Spezifikationsumgebung ist es daher wünschenswert, eine Spezifikationssprache zur Verfügung zu haben, die sowohl Algebren als auch Coalgebren unterstüzt. Es ist weiterhin von Vorteil, wenn die Spezifikationsumgebung *iterierte Spezifikationen* zulässt. Iterierte Spezifikationen sind Spezifikationen, in denen eine (co–)algebraische Spezifikation auf einen zuvor mittels algebraischer oder coalgebraischer Methoden definierten Typ Bezug nimmt. Kapitel 4 stellt eine Spezifikationssprache vor, die die eben genannten Bedinungen erfüllt: CCSL.

Für CCSL bestanden die folgenden Designziele:

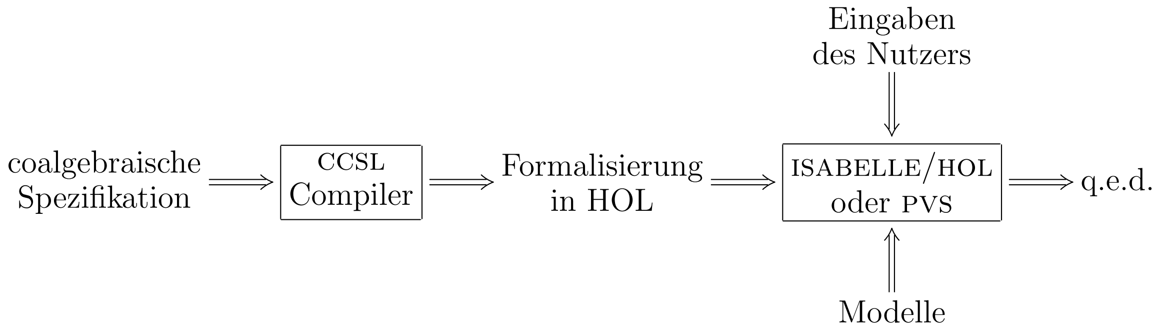
- Die Unterstützung von parametrisierbaren Klassenspezifikationen auf der Basis von Coalgebren.
- Die Unterstützung von algebraischen Spezifikationen abstrakter Datentypen auf der Basis initialer Algebren.
- Die Einbindung einer bekannten Logik.
- Möglichst keine Beschränkung der Ausdrucks– und Modellierungsstärke.
- Die Unterstützung von Theorembeweisern.

Diese Designziele sind natürlich zum Teil umstritten. Das zweitletzte Ziel impliziert zum Beispiel dass auch Signaturen, die zu Funktoren höherer Ordung korrespondieren, in CCSL zulässig sind. Auf der einen Seite erfordert eine höhere Ausdrucksstärke

[7] CCSL ist ein Akronym für *Coalgebraic Class Specification Language.*

der Spezifikationssprache eine größere Verantwortung der Benutzer. Der Benutzer[8] muss selbst darauf acht geben, Fehler, die durch die nicht gerechtfertigte Annahme vertrauter Eigenschaften enstehen, auszuschließen. Auf der anderen Seite gibt eine höhere Ausdrucksstärke dem Benutzer jedoch die Freiheit zugunsten flexiblerer Signaturen auf vertraute strukturelle Eigenschaften zu verzichten (oder umgekehrt). Insbesondere werden damit Experimente mit Signaturen möglich, für die im Moment nur wenig allgemeine theoretische Resultate zur Verfügung stehen. Die genannten Designziele werden in der Einführung zu Kapitel 4 ausführlicher erläutert.

In der typischen Anwendung wird CCSL zusammen mit einen Theorembeweiser wie folgt benutzt:



Spezifikationen in CCSL können mit Hilfe des CCSL Compilers in ihre Semantik in Logik höherer Ordnung übersetzt werden. Der CCSL Compiler unterstützt die beiden Theorembeweiser PVS und ISABELLE/HOL (in der Syntax von ISAR). Damit ist es möglich, die Eigenschaften einer CCSL Spezifikation in einem Theorembeweiser zu untersuchen. Der Nutzer kann im Theorembeweiser auch Modelle der CCSL Spezifkation entwickeln oder Verfeinerungen zwischen verschiedenen Spezifikationen konstruieren. Die starke Anbindung an Theorembeweiser macht CCSL zu einem nützlichen Werkzeug, obwohl es selbst noch Forschungsgegenstand ist. Die Spezifikationssprache CCSL wurde bereits in einer Reihe von Fallstudien verwendet, siehe Abschnitt 4.10.

Der wissenschaftliche Beitrag von Kapitel 4 besteht in der umfassenden Beschreibung der Spezifikationssprache CCSL, ihrer Syntax und ihrer Semantik in der Kategorie **Set** der Mengen und totalen Funktionen. Die Typtheorie von CCSL ist eine spezialisierte Version der polymorphen Typtheorie $\lambda{\to}$ aus (Barendregt, 1992; Jacobs, 1999a). Die Logik von CCSL ist eine Logik höherer Ordnung (Gordon and Melham, 1993) über dieser Typtheorie, die um Verhaltensgleichheit und modale Operatoren erweitert wurde. Die konkrete Syntax von CCSL ist stark an die Syntax von PVS angelehnt.

Spezifikationen in CCSL werden aus drei Bausteinen zusammengesetzt: Coalgebraischen Klassenspezifikationen, algebraischen abstrakten Datentypen und Signaturerweiterungen. Klassenspezifikationen bestehen aus Methodendeklarationen, Konstruktorde-

[8] Ich weise darauf hin, dass, wann immer im Text von einem „Benutzer“, „Leser“ oder ähnlichem die Rede ist, die Bezeichnung in der grammatisch maskulinen Form, dem allgemeinen Sprachgebrauch folgend, der an dieser Stelle nicht zur Debatte stehen kann, geschlechtsneutral verwendet wird.

klarationen und Axiomen, die das Verhalten von Methoden und Konstruktoren einschränken. Die Methodendeklarationen werden zu einer coalgebrischen Signatur zusammengefasst. Die zulässigen Signaturen entsprechen der Klasse der polynomialen Funktoren höherer Ordnung. Für abstrakte Datentypen gelten die üblichen Einschränkungen (Gunter, 1992; Berghofer and Wenzel, 1999), die die Existenz initialer Modelle sichern. Mit Hilfe von Signaturerweiterungen können in CCSL Typkonstruktoren, Konstanten und Funktionen deklariert werden, die im genutzten Theorembeweiser definiert sind.

Abbildung 2 zeigt als Beispiel die Spezifikation von (coalgebraischen) Warteschlangen in CCSL. Die ersten fünf Zeilen definieren den Typkonstruktor Lift.[9] Der Typkonstruktor Lift fügt zu seinem Argument ein zusätzliches Element bot hinzu und wird in CCSL zur Modellierung partieller Funktionen und Methoden benutzt.

Die Klassenspezifikation Queue deklariert einen Typparameter A für die Elemente der Warteschlange. Die Klassenspezifikation enthält zwei Methoden. Die Methode put fügt ein neues Element in die Warteschlange ein. Die Methode top entfernt das erste Element aus der Warteschlange und liefert als Ergebnis ein Paar, bestehend aus dem eben entfernten ersten Element und dem Folgezustand der Warteschlage. Für leere Warteschlagen schlägt top fehl und liefert bot als Ergebnis. Der Konstruktor new erlaubt es, neue Warteschlangen zu erzeugen.

Das Schlüsselwort **Assertion** leitet in Zeile 13 die Methodenaxiome ein. Das erste Axiom q_empty spezifiziert das Verhalten von leeren Warteschlagen. CCSL gestattet objektorientierte Notation für Methodenaufrufe: x.top steht für die Anwendung der Methode top auf die Warteschlagne x. Die Tilde ∼ bezeichnet Verhaltensgleichheit. Ihre Anwendung in Zeile 15 bedeutet, dass die Methode top, angewendet auf das Ergebnis von x.put(a), nicht fehlschlagen darf (das heißt, das Ergebis ist ungleich bot). Desweiteren muss das zurückgelieferte Paar das Element a enthalten und einen Zustand der zu x bisimilar ist.

Das Axiom q_filled legt das Verhalten für nichtleere Warteschlagen fest: Für nichtleere Warteschlangen kann man die Operationen put und top vertauschen. Schließlich legt das Konstruktoraxiom q_new fest, dass neu erzeugte Warteschlangen leer sind.

Es ist zu beachten, dass die Spezifikation Queue Modelle, die Warteschlangen mit unendlich vielen Elementen enthalten, *nicht* ausschließt.

Im letzten Teil der Spezifikation steht das Theorem strong_reachable. Theoreme beeinflussen die Semantik einer Klassenspezifikation nicht. Sie bieten aber eine bequeme Möglichkeit Sätze in der Syntax von CCSL unabhängig vom verwendeten Theorembeweiser zu formulieren. Das Theorem strong_reachable benutzt den modalen Operator **Eventually**. Es besagt, dass für alle Warteschlangen p und q mit endlich vielen Elementen ein Nachfolgezustand r von p existiert, der zu q bisimilar ist.

Der zur Dissertation gehörende Quellcode enthält (unter anderem) die Spezifika-

[9] Der Datentyp Lift entstammt dem CCSL Standard Vorspann (siehe Abschnitt 4.9.8) und wurde nur zur Information in Abbildung 2 eingefügt.

```
Begin Lift[ X : Type ] : Adt
   Constructor
      bot : Carrier;
      up( down ) : X –> Carrier
End Lift

Begin Queue[ A : Type ] : ClassSpec
  Method
    put : [Self, A] –> Self;
    top : Self –> Lift[[A,Self]];

  Constructor
    new : Self;

  Assertion Selfvar x : Self
    q_empty :  x.top ∼ bot      Implies
        Forall(a : A) . x.put(a).top ∼ up(a,x);

    q_filled :   Forall(a1 : A, y : Self) . x.top ∼ up(a1, y)      Implies
        Forall(a2 : A) . x.put(a2).top ∼ up(a1, y.put(a2));

  Creation
    q_new : new.top ∼ bot;

  Theorem
    Importing QueueModal[Self, A]

    strong_reachable :  Forall(p, q : Self) :
      Let finite? : [Self –> bool] = Lambda(q : Self) :
               (Eventually Lambda(x : Self) : x.top = bot For {top}) q
      IN
        finite? p And finite? q   Implies
          (Eventually
               (Eventually Lambda(r : Self) : r ∼ q For {put})
           For {top}
          ) p;
End Queue
```

Abbildung 2.: Spezifikation einer Warteschlange in CCSL

tion Queue, ein Modell der Spezifikation in PVS und einen Beweis für das Theorem strong_reachable.

Das Design von CCSL und die Implementierung des CCSL Compilers ist das Resultat einer Gruppenarbeit im LOOP Projekt. (LOOP ist ein Akronym für *Logic of Object-Oriented Programming.*[10]) Gegenstand des Projektes sind formale Methoden für objektorientierte Programmiersprachen. Das LOOP Projekt startete 1997 als gemeinsames Projekt an der Katholieke Universiteit Nijmegen (Universität Nimwegen) und der Technischen Universität Dresden. Außer mir arbeiten oder arbeiteten die folgenden Personen im LOOP Projekt: Joachim van den Berg, Ulrich Hensel, Jesse Hughes, Marieke Huisman, Bart Jacobs, Erik Poll und Jan Rothe. Innerhalb des Projekts werden verschiedene Forschungsrichtungen verfolgt. Die Gemeinsamkeit besteht in der Anwendung von Coalgebren als semantische Grundlage für objektorientierte Programme und der intensiven Nutzung von Theorembeweisern. Neben der Forschungsarbeit, die in der vorliegenden Dissertation ihren Niederschlag findet, ist die Semantik der Programmiersprache Java und die Verifikation von Java und Java Card Programmen Gegenstand der Forschung im LOOP Projekt (siehe (Huisman, 2001)). Die Arbeiten zu CCSL gehen bis zu den Anfängen des LOOP Projektes zurück. Alle Mitglieder im Projekt haben in der einen oder anderen Weise, oft substantiell, zu CCSL beigetragen.

Ein wichtiges Anwendungsgebiet von CCSL ist die Konstruktion von *Verfeinerungen.* Eine Verfeinerung setzt zwei Spezifikationen in Beziehung: Eine konkrete Spezifikation $\mathcal{C}$ verfeinert eine abstrakte Spezifikation $\mathcal{A}$, falls alle Modelle von $\mathcal{C}$ in Modelle von $\mathcal{A}$ überführt werden können. Für die Anwendung von Softwareverifikation in der Praxis ist der Begriff der Verfeinerung unverzichtbar.

In gemeinsamer Arbeit mit Bart Jacobs habe ich parallel zur Entwicklung von CCSL auch eine Definition für *coalgebraische Verfeinerung*, das heißt, für Verfeinerung zwischen coalgebraischen Spezifikationen, ausgearbeitet. Eine ausführliche Darstellung ist in (Jacobs and Tews, 2001). In der vorliegenden Arbeit enthält Abschnitt 4.10.1 eine kurze Beschreibung coalgebraischer Verfeinerung. In diesem Abschnitt wird auch gezeigt, wie Verfeinerungen zwischen CCSL Spezifikationen bewiesen werden können.

Verifikation der Doktorarbeit

Eine Besonderheit der vorliegenden Arbeit ist ihre enge Beziehung zum Theorembeweiser PVS. Diese Arbeit beschreibt nicht nur Softwarewerkzeuge, die in Verbindung mit PVS zur Softwareverifikation genutzt werden können. Auch die meisten theoretischen Ergebnisse und fast alle Beispiele dieser Arbeit wurden mit PVS entwickelt, beziehungsweise überprüft. Dazu wurde ein Teil der in dieser Arbeit benutzten Kategorientheorie in PVS formalisiert. Die dieser Formalisierung zugrunde liegenden Ideen werden im Abschnitt 2.4.4 erläutert. Weitere Details und Ausschnitte aus dem PVS Quellcode enthält

[10] Im WWW ist das LOOP Projekt unter der URL http://www.cs.kun.nl/~bart/LOOP/ zu finden.

Anhang A.1. Propositionen und Lemmata, die eine direkte Entsprechung zu einem Theorem in PVS haben (und somit in PVS bewiesen wurden), können in dieser Arbeit am Satz „This lemma/proposition has been proved in PVS.“ erkannt werden. Abschnitt A.2 des Anhangs enthält eine Tabelle, die Propositionen, Lemmata und Beispiele in dieser Arbeit mit dem PVS Quelltext in Beziehung setzt. Der PVS Quelltext ist im WWW verfügbar, siehe Anhang A.

Während der gesamten Zeit, in der ich mich mit dieser Doktorarbeit beschäftigt habe, erwies sich PVS als ausgezeichnetes Werkzeug um Beispiele zu entwickeln, Ideen zu prüfen und natürlich um Lemmata zu beweisen. Allerdings gibt es auch eine zweite Seite von PVS: Die riesige Anzahl an Fehlerberichten, die ich im Verlauf dieser Arbeit in der PVS Fehlerdatenbank eingereicht habe, zeigen, dass PVS, als Softwaresystem betrachtet, noch enorme Entwicklungsmöglichkeiten hat.

Verwandte Forschungsarbeiten

Die vorliegende Arbeit basiert auf älteren Arbeiten zum Thema Coalgebren und coalgebraischer Spezifikation. Die wesentliche Motivation dieser Arbeit ist die Idee von (Reichel, 1995), Coalgebren als Grundlage der Semantik objektorientierter Sprachen zu verwenden. Diese Idee wird in mehreren Arbeiten (Jacobs, 1995; Jacobs, 1996a; Jacobs, 1996b; Jacobs, 1997a; Jacobs, 1997b) von Jacobs weitergeführt. Die Beziehung zwischen algebraischen und coalgebraischen Spezifikationen wird in (Hensel and Jacobs, 1997; Hensel, 1999; Rößiger, 2000a; Rößiger, 2000b) untersucht. Hensel und Jacobs entwickeln in diesen Arbeiten hinreichende Kriterien für die Gültigkeit von Induktions– und Coinduktionsprinzipien. Rößiger beweist die Existenz von initialen Algebren und finalen Coalgebren in der Kategorie der Mengen für alle kovarianten Datenfunktoren. Beide Ergebnisse bilden das Fundament der Semantik von CCSL.

Coalgebren sind gegenwärtig ein aktives Forschungsgebiet, siehe (Jacobs et al., 1998b; Jacobs and Rutten, 1999; Reichel, 2000; Corradini et al., 2001). Das Problem der Behandlung binärer Methoden ist schon längere Zeit bekannt, eine allgemeine Lösung existierte jedoch nicht. Jacobs erläutert in (Jacobs, 1996a) wie man das Problem umgehen kann indem man die binären Methoden als definitorische Erweiterungen behandelt. Eine weitere (Teil–) Lösung schlagen Hennicker und Kurz in (Hennicker and Kurz, 1999) vor: Unter gewissen Voraussetzungen können binäre Methoden mit einem Wertebereich von Self als algebraische Erweiterungen coalgebraischer Spezifikationen formalisiert werden.

Die Wurzeln der Spezifikationssprache CCSL reichen bis zu den ersten, oben zitierten Arbeiten von Jacobs zurück. In diesen Arbeiten betrachtet Jacobs coalgebraische Signaturen als spezielle polymorphe Signaturen. Der einfachste Weg eine coalgebraische Logik[11] zu erhalten, ist demzufolge, eine bekannte Logik (zum Beispiel Gleichungslo-

[11] In dieser Arbeit verwende ich die Phrase „coalgebraische Logik“ für alle Logiken, die man mit coalgebraischen Signaturen einsetzen kann. Die in (Moss, 1999) beschriebene *coalgebraic logic* ist somit eine spezielle coalgebraische Logik.

gik) für polymorphe Signaturen zu verwenden. Dieser Ansatz wurde auch im Design von CCSL verfolgt. Goldblatt beschreibt in (Goldblatt, 2001a; Goldblatt, 2001b) das Fragment erster Stufe der Logik von CCSL. In dieser Arbeit entwickelt Goldblatt auch ein Birkhoff–artiges Theorem. Die modalen Operatoren mit Methodengranularität von CCSL stammen aus (Rothe, 2000). Die allgemeinere Version mit Pfadgranularität wird in (Jacobs, 1999b) entwickelt.

Die Spezifikationsprache CCSL ist sehr eng mit der Programmiersprache Charity (Cockett and Fukushima, 1992; Schroeder, 1997) verwandt. In Charity kann man nur mit Hilfe von initialen Algebren oder finalen Coalgebren programmieren. Die These, dass CCSL die perfekte Spezifikationssprache für Charity Programme sei, ist also durchaus gerechtfertigt.

In der Forschung zum Thema Coalgebren wurden schon viele Ansätze für eine coalgebraische Logik vorgestellt. Einige Autoren verfolgen die Idee, dass eine Logik für Coalgebren (also dualisierten Algebren) auf dualisierten Gleichungen beruhen muss. Solche *Cogleichungen* werden zum Beispiel in (Cîrstea, 1999) vorgestellt. In dieser Arbeit entwickelt Cîrstea einen korrekten und vollständigen Deduktionskalkül für eine eingeschränkte Menge coalgebraischer Signaturen. Binäre Methoden sind in den Signaturen von Cîrstea nicht zugelassen. In der vorliegenden Arbeit wird dem Problem der Vollständigkeit (des Deduktionskalküls) keine große Wertigkeit eingeräumt. Ich erachte eine ausreichende Ausdrucksstärke, die es gestattet, echte Anwendungsfälle bequem zu behandeln, für wesentlich wichtiger.

Eine Reihe von Arbeiten analysiert das Verhältnis von Coalgebren und modaler Logik, zum Beispiel (Moss, 1999; Rößiger, 2000a; Kurz, 2000; Hughes, 2001). Diese genannten Arbeiten beschreiben unterschiedliche modale Logiken. Moss entwickelt charakterisierende Formeln, das heißt Formeln, mit denen Zustände eindeutig bis auf Verhaltensgleichheit beschrieben werden können. Rößiger erhält einen vollständigen Deduktionskalkül für seine Logik und konstruiert mit Hilfe der Logik finale Coalgebren. Kurz und Hughes benutzen modale Logik in ihren Arbeiten nach der Suche eines Birkhoff–artigen Satzes. Die Entwicklung all dieser verschiedenen modalen Logiken wurde immer von theoretischen Fragestellungen begleitet und beeinflußt. Als Ergebis dessen sind diese Logiken zur Spezifikation von Systemen relativ ungeeignet. In der Logik Rößigers benötigt man zum Beispiel für die einfache Aussage, dass eine Warteschlange nur endlich viele Elemente enthält, eine unendliche Menge von modalen Formeln. Aus diesem Grund spielten die eben diskutierten modalen Logiken bei der Gestaltung der Logik von CCSL nur eine geringe Rolle.

Verborgene Universelle Algebra oder Universelle Algebra mit verborgenen Sorten (englisch *hidden algebra*) (Roşu, 2000; Goguen and Malcolm, 2000) ist ein Zweig mehrsortiger Universeller Algebra, in dem einige Sorten als verboren oder versteckt betrachtet werden. Auf verborgenen Sorten gibt es keinen direkten Zugriff. Sie sind dafür gedacht, den Zustansraum von Automaten oder Klassen zu modellieren. Eine erhebliche Einschränkung in der Arbeit mit verborgenen Algebren besteht darin, dass Signaturen nur algebraische Operationen der Form $S_1 \times \cdots \times S_n \longrightarrow S_0$ enhalten können, wobei alle S_i

Sorten bezeichnen. Das heißt, es gibt weder strukturierte Argumente noch strukturierte Ergebnistypen. Mit Coalgebren kann Partialität sehr einfach mit Operationen der Form $\mathsf{Self} \longrightarrow \mathsf{Self} + \mathbf{1}$ modelliert werden. Im Gebiet verborgener Algebren ist es hierfür notwendig, Teilsorten zu verwenden.

Ein weiterer Unterschied im Vergleich mit verborgenen Algebren ist der unterschiedliche Ansatz Verhaltensgleichheit zu beschreiben. Für coalgebraische Spezifikationen verwendet man Bisimulationen, ein Begriff, mit dem man auch die Verhaltensgleichheit von Zuständen verschiedener Coalgebren untersuchen kann. Im Gebiet der verborgenen Algebren wird der Ansatz der sichtbaren Kontexte von (Reichel, 1985) verwendet, um sogenannte verborgene Kongruenzen (englisch *hidden congruence*) zu definieren. Sobald binäre Methoden eine Rolle spielen, kann man mit verborgenen Kongruenzen nur noch Zustände eines Modells behandeln.

Abschließend möchte ich noch drei andere Systeme zur Softwarespezifikation betrachten. Die Arbeit (Kellomäki, 1997) beschreibt die Spezifiktionssprache DisCo für reaktive Systeme. DisCo basiert auf der temporalen Logik von Aktionen (TLA) von Lamport (Lamport, 1994). In DisCo beschreibt man Klassen nur durch ihre Datenfelder. Außerhalb der Klassen werden die möglichen Aktionen, die im System auftreten können, zusammen mit den Zustandsänderungen beschrieben.

Die Initiative COFI (*common framework initiative*) (Mosses, 1997) entwickelt die Spezifikationssprache CASL (*Common Algebraic Specification Language*[12]). Für CASL sind eine Reihe von Fragmenten definiert, die den verschiedenen Logiken entsprechen, die im Gebiet der algebraischen Spezifikation benutzt wurden. In CASL gibt es jedoch keine Möglichkeit, Verhalten oder Prozesstypen zu beschreiben.

Die Modellierungssprache UML (Fowler, 1999; OMG, 2001) zielt zusammen mit der speziellen Logik OCL (Warmer and Kleppe, 1999; OMG, 1997) genau wie CCSL auf die Beschreibung objektorientierter Software. Während CCSL jedoch eine Spezifikationssprache ist, handelt es sich bei UML/OCL hauptsächlich um ein Entwurfswerkzeug. Ein Vergleich zwischen CCSL und UML/OCL ist gegenwärtig nicht möglich, da am UML Standard noch intensiv gearbeitet wird (Kobryn, 1999) und der gegenwärtige Standard zur Semantik von UML/OCL nur sehr wage Aussagen macht. Die beispielhafte Übertragung eines UML/OCL Beispieles in Abschnitt 4.10.3 unterstützt die These, dass die meisten Konstrukte von UML Klassendiagrammen sowie von OCL problemlos in CCSL modelliert werden können. Eine Einbettung von CCSL in den formalen Teil von UML/OCL ist jedoch unmöglich. Das liegt zum einen an der Ausdrucksstärke von OCL (die der von Aussagenlogik entspricht). Zum anderen lässt UML nur die Spezifikation von Klassen zu. Es gibt keine Unterstützung für algebraische Datentypen.

[12] Nicht zu verwechseln mit der *custom attack simulation language* (CASL) (Vigna et al., 2000; Secure Networks, 1998).

1. Introduction

This thesis is about *coalgebraic methods* in *software specification and verification.* It extends known techniques of coalgebraic specification to a more general level to pave the way for real world applications of software verification.

There are two main contributions of the present thesis:

- Chapter 3 proposes a generalisation of the familiar notion of coalgebra such that classes containing methods with arbitrary types (including binary methods) can be modelled with these generalised coalgebras.

- Chapter 4 presents the specification language CCSL (short for Coalgebraic Class Specification Language), its syntax, its semantics, and a prototype compiler that translates CCSL into higher-order logic.

In flavour coalgebraic specification as presented in this thesis is very similar to algebraic specification. It builds on signatures consisting of coalgebraic operations and uses (a variant of) equational logic to restrict the class of models. However, coalgebraic specification is not to replace algebraic specification. The aim is rather to combine both algebraic and coalgebraic specification techniques in one specification environment to allow the user to specify parts of his system algebraically while other parts are modelled coalgebraically. The specification language CCSL in combination with the CCSL compiler and one of the theorem provers ISABELLE/HOL or PVS provides such an environment. Through the combination of the advantages of algebraic and coalgebraic techniques the complexity of the overall specification can be drastically reduced. This makes the application of formal methods cheaper or moves systems into the scope of formal methods that are too complex for traditional methods of software verification.

Let me explain some of the buzzwords before I continue with a more detailed introduction into this thesis. An *algebra* is a function with a *structured domain* like

$$\mathsf{cons} : \ A \times \mathsf{List}[A] \longrightarrow \mathsf{List}[A]$$

An *algebraic signature* is a finite set of such algebra declarations. An *algebraic specification* consists of an algebraic signature and a set of axioms. A *model* of such a specification is a set of functions that fulfils the axioms. In algebraic specification one models pieces of software as algebras and describes (and proves) their properties within a logical framework. With algebraic specification one can very nicely describe the properties of finitely generated data structures (like lists or binary trees) and functions (i.e., programs) that manipulate these data. Thereby the description of the data structures takes place at an abstract level, without referring to a concrete implementation.

Verification means the development of formal proofs that show that the modelled software behaves as intended. This may involve the derivation of properties from the axioms of a specification, the proof of refinement relations between different specifications, and the construction of models. The mathematics that is involved in software verification if often rather simple. However, a complete verification requires many case distinctions and a huge amount of simple reasoning and computation. Therefore non-trivial verification examples require the use of *theorem provers*. A theorem prover is a software tool, that implements a logic and a derivation system to check and find proofs. A theorem prover can do all the simple reasoning and take care of the bureaucratic tasks in a verification (i.e., checking side conditions, ensuring that case distinctions are complete, and so on). The person who carries out the verification can then concentrate on the important (and difficult) parts. In the present thesis I consider the theorem provers PVS (Owre et al., 1996) and ISABELLE/HOL (Nipkow et al., 2002b; Nipkow et al., 2002a). Both tools are interactive theorem provers for higher-order logic.

A *coalgebra* is a function with a *structured codomain* like[1]

$$\mathsf{step} : \ \mathsf{Aut}[A, B] \longrightarrow A \Rightarrow (\mathsf{Aut}[A, B] \times B)$$

or equivalently[2]

$$\mathsf{step}' : \ \mathsf{Aut}[A, B] \times A \longrightarrow \mathsf{Aut}[A, B] \times B$$

If you consider $\mathsf{Aut}[A, B]$ as the set of (not necessarily finite) automata with input A and output B then step (or step') is the transition function that maps any state of an automaton together with an input from A to a successor state and an output in B. The set $\mathsf{Aut}[A, B]$ is the *state space* of the coalgebra. Its elements —the states— are considered as black boxes, that is you cannot investigate their internal structure. You can only apply the coalgebra step, provide additional input from A and make an *observation* in B. You can continue to apply the coalgebra step to the successor state to obtain more observations. Note that this process can go on for ever, so coalgebras can very naturally model nonterminating, infinitely running processes.

The set of all observations that one can make about a given element $x \in \mathsf{Aut}[A, B]$ by providing all possible inputs and inspecting all successor states in the same way is called the *observable behaviour* of x. It is possible that two different states show the same observable behaviour, in this case they are *observably equivalent*. The notion of *bisimulation* captures observational equivalence formally. A bisimulation is a relation on the state space of a coalgebra which relates states of the same observable behaviour and

[1] Here $A \Rightarrow (\mathsf{Aut}[A, B] \times B)$ denotes the set of functions $A \longrightarrow \mathsf{Aut}[A, B] \times B$, which is often written as $(\mathsf{Aut}[A, B] \times B)^A$.

[2] Recall that functions $X \longrightarrow Y \Rightarrow Z$ are in one–to–one correspondence with functions $X \times Y \longrightarrow Z$.

which is closed under taking successor states. In the application domain of coalgebras one is mainly interested in the question if two systems (or two states) behave in the same way. If they are equal is usually not of interest. Therefore, for coalgebras, the notion of bisimulation replaces equality both in specifications and in reasoning.

Coalgebras can easily model exceptional or partial behaviour. Consider[3]

$$\mathsf{step}_p : \mathsf{PAut}[A, B] \longrightarrow A \Rightarrow (\mathsf{PAut}[A, B] \times B \;+\; \mathbf{1})$$

or equivalently

$$\mathsf{step}'_p : \mathsf{PAut}[A, B] \times A \longrightarrow \mathsf{PAut}[A, B] \times B \;+\; \mathbf{1}$$

Elements of $\mathsf{PAut}[A, B]$ are (possibly infinite) partial automata that might fail on an input, producing $* \in \mathbf{1}$ instead of a successor state and an output in B.

A *coalgebraic (class) specification* consists of a finite set of coalgebra declarations —the *coalgebraic (class) signature*— and a finite set of axioms. The axioms are formulated in a logic to be made precise in Chapter 4. Typical axioms require certain behaviour, for instance[4]

$$\forall x \in \mathsf{PAut}[A, B] \;.\quad \mathsf{step}_p\, x\, a_0 \;\neq\; *$$

requires that no automaton might fail for a special input $a_0 \in A$. Another typical form of axioms requires that certain states are behaviourally equivalent. Here is an axiom that requires all automata to behave cyclicly with cycle length two:

$$\forall x \in \mathsf{Aut}[A, B] \;.\; \forall a_0, a_1 \in A \;.\quad x \;\underline{\leftrightarrow}\; \pi_1\big(\mathsf{step}\; (\pi_1\; (\mathsf{step}\, x\, a_0))\;\; a_1\big)$$

(The relation $\underline{\leftrightarrow}$ denotes the greatest bisimulation and $\pi_1 : \mathsf{Aut}[A, B] \times B \longrightarrow \mathsf{Aut}[A, B]$ is the first projection.)

Besides bisimulation there are two other important notions for coalgebras: *coalgebra morphisms* and *invariants*. A coalgebra morphism is a structure preserving function between the state spaces of two coalgebras (for the same signature). An invariant is a property (of the states of a coalgebra) that, once it holds for a state x, continues to hold for all direct or indirect successor states of x. Invariants play an important role in defining logics for coalgebraic specifications and in coalgebraic refinement (see (Jacobs and Tews, 2001) for a presentation of coalgebraic refinement).

A popular approach is to define both bisimulations and invariants in terms of coalgebra morphisms. The definition of coalgebra morphisms relays only on the structure of the signature.

[3] Here $+$ describes the disjoint union and $\mathbf{1} = \{*\}$ the one element set.

[4] Following the HOL tradition I write function application without explicit parenthesis. Function application associates to the left, so $\mathsf{step}_p\, x\, a_0 = (\mathsf{step}_p(x))(a_0)$.

The present thesis pursues the idea of using algebras and coalgebras to model software. In general, programming languages provide a framework quite different from set theory. Different in the sense that constructions that are possible in set theory are not possible in the programming language (and sometimes vice versa). Similarly for abstract properties that do or do not hold. For instance, for any two sets A and B one can build the set $A \Rightarrow B$ of functions from A to B. In contrast most programming language do not allow the formation of function types. Moreover the programming languages themselves differ.

This suggests to work in a more abstract setting than set theory in which one has more control over the available constructions and properties. *Category theory* provides such a framework. A category can be viewed as an abstraction of an universe of sets. It contains objects (the abstraction of sets) and morphisms (the abstraction of functions). In general, a category possesses very little structure. For instance, in order to be able to talk about pairs one has to assume the existence of *products* — a certain structure in the category. This way one can tailor a category to match the properties of a given programming language very precisely.

When working in an arbitrary category $\mathbb{C}$ *endofunctors* $F : \mathbb{C} \longrightarrow \mathbb{C}$ play the role of signatures. (An endofunctor is a mapping from the category onto itself, which is required to preserve certain structure.) A coalgebra is then simply a morphism $X \longrightarrow F(X)$, where X —the state space— is an object from $\mathbb{C}$. Similarly an algebra is a morphism $F(X) \longrightarrow X$, where F models the algebraic signature. The simple structure of types in most programming languages implies that also the signatures that one wants to model have a simple structure. This has further implications about the class of endofunctors that one has to consider. For many applications it is sufficient to consider only *polynomial functors*. Polynomial functors are finitely generated from constants (for constant types like the booleans), products (for record types), coproducts (for variant records or union), and exponents (for function types) (polynomial functors are defined in Subsection 2.6.1).

With respect to the logic that is used to reason about specifications and programs there is a similar argument about the choice of the abstraction level: Different theorem provers provide logics with slightly different properties. So it is better to work in an abstract logical setting. The theory of *fibrations* provides a generalisation of logic within category theory.

For these reasons the main chapters of the present thesis rely in a nontrivial way on category theory. The preliminary Chapter 2 provides all the definitions and explains all the notation that is beyond simple set theory. In particular the first three sections of Chapter 2 introduce the essential definitions from category theory and fibred category theory. Section 2.4 presents two fibrations in detail: The fibration of (typed) predicates and that of (typed) relations. Both fibrations and the properties collected in Section 2.4 are essential for Chapter 3. Section 2.5 and Section 2.6 introduce algebras and coalgebras. The section on coalgebras contains a collection of results about coalgebras for polynomial functors that have been collected from the literature. These results are important for Chapter 3, where I consider generalisations of polynomial functors.

Binary Methods

Chapter 3 elaborates on the problem of how to formalise arbitrary method types, including *binary methods*, with coalgebras. The term *method* comes from object-oriented programming. There the software is organised in *classes*, which represent data abstractions and operations. The runtime structures of classes are *objects*. The operations are called *methods*. Every method is associated to one class. The primary computational model is method invocation, typically written as $o.f$, where o is an object of some class with a method f. On invocation the method f gets o as an implicit argument and can therefore access the data in o. Typically the internal structure of an object is not visible from the outside (hiding implementation details). If one wants to access the data of an object one has to use the publicly available methods of its class.

The information hiding aspect and the computational pattern of object orientation suggest to use coalgebras as a semantic foundation of object orientation. This idea was first proposed in (Reichel, 1995). There was, however, one deficiency: Classes that contain binary methods cannot be modelled as coalgebras (for endofunctors). Assume that we enrich the interface of the automata with a merge method that takes two automata a and b as arguments and returns their interleaving (which invokes a on the first input to compute the result, b on the second, and so on). The type of merge is obviously

$$\mathsf{merge}: \ \mathsf{Aut}[A, B] \times \mathsf{Aut}[A, B] \longrightarrow \mathsf{Aut}[A, B]$$

By exploiting the fact that every function $X \times Y \longrightarrow Z$ corresponds uniquely to a function $X \longrightarrow Y \Rightarrow Z$ we can present merge in coalgebraic shape:

$$\mathsf{merge}: \ \mathsf{Aut}[A, B] \longrightarrow \mathsf{Aut}[A, B] \Rightarrow \mathsf{Aut}[A, B]$$

Now the state space $\mathsf{Aut}[A, B]$ occurs at a contravariant position (at the left hand side of $\Rightarrow$) in the codomain of merge. Therefore, the automaton signature with the merge operation cannot be modelled with a polynomial functor.[5]

Chapter 3 presents a solution to the problem of binary methods by using bivariant functors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ to model signatures and an appropriate notion of coalgebra. To judge if the proposed generalisation makes sense and can be used with benefit I use the following criterion: First, there must exist sensible definitions of the notions of coalgebra morphism, bisimulation, and invariant. Second, these notions should have similar properties like they have for polynomial functors. To facilitate this task the introductory Section 2.6 on coalgebras lists more than ten results about coalgebras for polynomial functors (for instance the existence of the final coalgebra, that bisimulations and invariants form complete lattices and so on).

The generalised definition for coalgebra and coalgebra morphisms is given in Section 3.2 (Definition 3.2.2 on page 80). The definition of bisimulation and invariant is in

[5] More precisely, the coalgebraic signature of merge can be modelled with an endofunctor only on a category in which every morphism is reversible.

Class of Functors	Property	Example
polynomial	constant arguments	$\mathsf{Self} \times A \longrightarrow \boxed{\cdots}$
extended cartesian	no functional arguments	$\mathsf{Self} \times \mathsf{Self} \times \mathsf{Self} \longrightarrow \boxed{\cdots}$ $\mathsf{Self} \times (\mathsf{Self} + A) \longrightarrow \boxed{\cdots}$
extended polynomial	functional arguments with constant domain	$\mathsf{Self} \times (A \Rightarrow \mathsf{Self}) \longrightarrow \boxed{\cdots}$
higher-order polynomial	arbitrary arguments	$\mathsf{Self} \times (\mathsf{Self} \Rightarrow A) \longrightarrow \boxed{\cdots}$ $\mathsf{Self} \times (\mathsf{Self} \Rightarrow \mathsf{Self}) \longrightarrow \boxed{\cdots}$

Table 1.1.: Classes of Functors investigated in Chapter 3

Section 3.3 (Definition 3.3.3 on page 85). Thereby I follow the approach of (Hermida and Jacobs, 1998) and exploit predicate and relation lifting. A careful investigation of the properties of these notions yields three different levels of generalisations of polynomial functors. With an increasing level of generality less properties hold in general. See Table 1.1 for an overview.

The most general class of functors discussed in the present thesis is the class of *higher-order polynomial functors*. It is investigated in Section 3.3. For coalgebras of higher-order polynomial functors only trivial properties hold (like that the equality relation is a bisimulation). Everything else fails (see Fact 3.3.7 on page 88). For instance Example 3.3.8 shows a coalgebra and two bisimulations for it, such that the intersection of the two bisimulations is not a bisimulation. The examples that exhibit all these failures are surprisingly simple: The state space of the coalgebra in the counter examples contains always less than six elements. Another negative result about coalgebras for higher-order polynomial functors is that the traditional approach to define bisimulation via coalgebra morphisms following (Aczel and Mendler, 1989) fails: it yields a notion of bisimulation that is not closed under taking successor states.

The class of *extended polynomial functors* is a proper subclass of higher-order polynomial functors. It restricts the use of function types in arguments for methods: For method arguments of functional type $\sigma \Rightarrow \tau$ the type σ must be a constant, see Table 1.1 for an example. The precise definition is in Section 3.4. For extended polynomial functors most of the familiar properties hold. For instance bisimulations are closed under intersection and coalgebra morphisms are functional bisimulations. Coalgebras for extended polynomial functors have the following deficiencies: First, bisimulations and invariants are not closed under taking their union. Second, there is no final coalgebra (see Section 3.6).

The negative result about unions of invariants and and bisimulations can be improved by strengthening the definitions. In Subsection 3.4.6 I show that *strong invariants* form a complete lattice. With a stronger definition of the notion of bisimulation the bisimulations for one coalgebra form a (incomplete) lattice, see Subsection 3.4.7. Recent work (Tews, 2002b) shows that this lattice is complete (i.e., a greatest bisimulation exists) if the coalgebra is a computable function (in the intuitive sense).

Section 3.5 introduces the least generalisation of polynomial functors: The class of *extended cartesian functors* (see Definition 3.5.1 on page 108). Extended cartesian functors do not allow arguments of functional type. They are a subclass of extended polynomial functors, therefore coalgebras of extended cartesian functors have all the properties of coalgebras of extended polynomial functors. For extended cartesian functors it is possible to adopt a result from (Poll and Zwanenburg, 2001): bisimulations that are partial equivalence relations (on a fixed domain) form a complete lattice (see Theorem 3.5.9 on page 113). So for extended cartesian functors bisimulations that are equivalence relations are closed under union. However, also for extended cartesian functors there exist no final coalgebra in general.

A substantial part of Chapter 3 appeared previously as (Tews, 2000b; Tews, 2001; Tews, 2002b).

The Coalgebraic Class Specification Language CCSL

Both algebras and coalgebras have their advantages and disadvantages. Coalgebras are good for representing (possibly) infinitely running processes, algebras are good for finitely generated data types like lists or trees. Software systems usually involve both — data types and processes. For the specification it is therefore desirable to use a language that allows both algebraic and coalgebraic specifications. The language should further allow the nested use of all specifications. That is, it should allow (co)algebraic specifications that use types that have been defined by an algebraic or coalgebraic specification before. Such nested specifications are called *iterated specifications* in this thesis. Chapter 4 describes a language that has these properties: The *Coalgebraic Class Specification Language* CCSL.

The design goals of CCSL are:

- to provide a notation for parametrised class specifications based on coalgebras;
- to provide algebraic specifications of abstract data types based on initial algebras;
- to use a familiar logic;
- to restrict expressiveness only when absolutely necessary;
- to provide theorem proving support.

Of course, these design goals are arguable. For instance the second last item implies that even signatures that correspond to higher-order polynomial functors are accepted in CCSL. On the one hand, the (almost) unrestricted expressivity shifts much responsibility to the user of CCSL. He has to be careful not to introduce errors by assuming familiar properties that do not hold for the generalisation level he chose. On the other hand the user can himself decide if he wants to trade off general properties for more expressiveness (or vice verse). In particular one can experiment with signatures for which not much is known at the moment. The introduction to Chapter 4 discusses the preceding design goals and their implications further.

Specifications in CCSL can be translated into (their semantics in) the higher-order logic of the theorem provers PVS and ISABELLE/HOL (in new style ISAR syntax). This makes it possible to examine the properties of a CCSL specification in the theorem prover. One can, for instance, construct models in the theorem prover and check if they satisfy the specification. CCSL has already been applied successfully in a number of case studies, see Section 4.10.

The main contribution of Chapter 4 is the presentation of the specification language CCSL, its syntax, and its semantics as a whole. Most sections of Chapter 4 contain —when considered alone— only little original material. For instance the type theory of CCSL is a specialised version of the polymorphic type theory $\lambda\!\!\to$ from (Barendregt, 1992; Jacobs, 1999a). The logic of CCSL is a standard higher-order logic over this type theory. However, the combination of standard results from several areas of theoretical computer science yields CCSL as a specification language that can express finitely generated data types and infinite dynamic behaviour equally well.

The design and implementation of CCSL was a group effort in the LOOP project. LOOP stands for Logic of Object-Oriented Programming.[6] It is a project on formal methods for object-oriented languages. It started in 1997 as a joint project between the Katholieke Universiteit Nijmegen (University of Nijmegen) and the Technische Universität Dresden (Dresden University of Technology). Apart from myself the following people do or have been working within the LOOP project: Joachim van den Berg, Ulrich Hensel, Marieke Huisman, Bart Jacobs, Erik Poll, and Jan Rothe. There is a certain diversity in the research done in the LOOP project. The common underlying base is the use of coalgebras as a semantics for object orientation and the use of theorem proving support. Apart from the work that is described in the present thesis, the research in the LOOP project focuses on a formal semantics of the programming language Java and the verification of Java programs (see (Huisman, 2001)), especially for Java Card programs. Another topic is the design of JML, the Java Modelling Language (see (Leavens et al., 2000)). JML is an extension of Java that allows one to specify the detailed design of Java classes and interfaces. The work on CCSL goes back to the beginning of the LOOP project. All LOOP project members have contributed to CCSL in one or the other way, often substantially.

One primary application of the CCSL specification environment is the construction

[6] The LOOP project is on the world wide web, see URL http://www.cs.kun.nl/∼bart/LOOP/.

of *refinements*. Refinement is a relation between specifications: A concrete specification $\mathcal{C}$ refines an abstract specification $\mathcal{A}$ if all models of $\mathcal{C}$ satisfy (via a signature translation) also $\mathcal{A}$. The use of refinements is essential for real world applications of software specification and verification.

In joint work with Bart Jacobs we devised a notion of *coalgebraic refinement* in parallel with the development of CCSL. For several reasons the work on refinement is not included in the present thesis. An elaborated presentation is in (Jacobs and Tews, 2001). Subsection 4.10.1 contains a short description of coalgebraic refinement and explains how to prove refinements when working with CCSL.

The presentation in Chapter 4 concentrates on the less well known and more important aspects ofCCSL. The first section introduces the type theory of CCSL and the second section presents a formal notion of *variances*. Variances allow one to classify types and signatures into those that correspond to polynomial functors, to extended polynomial functors, and to higher-order polynomial functors. Section 4.4 and Section 4.5 present the coalgebraic part of CCSL. Section 4.6 introduces abstract data types and Section 4.7 goes into the semantics of iterated specifications. The more informal Section 4.8 discusses the relation of CCSL with important aspects of object-oriented programming like inheritance and late binding. Section 4.9 wraps the description of CCSL up and the following Section 4.10 presents applications of CCSL: Coalgebraic refinement, case studies that have been done with CCSL, and a translation of an UML example.

Verifying the Thesis

A special feature of the present thesis is its close relationship with the theorem prover PVS: The thesis does not only describe tools that make it possible to employ PVS for software verification. In addition, most of the theoretical results and examples that are presented here have been developed and checked with PVS. The main ideas of the formalisation that I use are laid out in Subsection 2.4.4. The Appendix A.1 shows some examples of the PVS code. Propositions and lemmas that have a direct counterpart in PVS can be recognised by the sentence “This lemma/proposition has been proved in PVS”, that appears in the proof. The complete PVS sources are available in the world wide web, see Appendix A for the details. The Appendix A.2 contains a table that relates the propositions, lemmas, and examples of this thesis with the PVS sources.

Throughout the work on this thesis PVS proved to be an excellent tool for developing and checking examples and ideas, and, of course, for proving lemmas. However, as usual for close friendships, I also got to know the less bright shining sides of PVS. The area were PVS has most development potential is stability and robustness as a software system. The huge number of submitted bug reports (almost all of them describe problems in the user interface and not inconsistencies in the logic) witnesses this.

Related work

The present thesis builds on earlier work on coalgebras and on coalgebraic specification. The main motivations for this thesis is the idea of (Reichel, 1995) to use coalgebras as a semantic foundation of object orientation. In a series of papers (Jacobs, 1995; Jacobs, 1996a; Jacobs, 1996b; Jacobs, 1997a; Jacobs, 1997b) Jacobs develops this idea further and builds the basis of what I call *coalgebraic specification*. A first investigation of the relation of algebraic specification and coalgebraic specification is in (Hensel and Jacobs, 1997; Hensel, 1999; Rößiger, 2000a; Rößiger, 2000b). In this work Hensel and Jacobs develop sufficient criteria for the validity of iterated induction and coinduction principles. Rößiger proves the existence of initial algebras and final coalgebras for all iterated functors in the category of sets and total functions. This shows that the carrier sets for iterated specifications do exist. All this forms the basis of the present thesis.

Currently coalgebras are an active research field, see (Jacobs et al., 1998b; Jacobs and Rutten, 1999; Reichel, 2000; Corradini et al., 2001). The problem of binary methods has been know for some time but I am not aware of any other rigorous solution. Jacobs shows in (Jacobs, 1996a) that one can sometimes avoid the problem by using definitional extensions. Another partial solution is suggested in (Hennicker and Kurz, 1999): Binary methods with codomain Self can sometimes be formalised as algebraic extensions.

The roots of the specification language CCSL can be traced back to the early work of Jacobs cited above. In this work Jacobs considers coalgebraic signatures as (special) polymorphic signatures. The easiest way to obtain a coalgebraic logic[7] is thus to use a well-known logic (such as equational logic) over coalgebraic signatures. This is the approach of CCSL. Goldblatt describes in (Goldblatt, 2001a; Goldblatt, 2001b) a first-order fragment of the logic of CCSL and develops a Birkhoff like theorem for it. The method-wise modal operators of CCSL come from (Rothe, 2000). Their more general path-wise version is studied in (Jacobs, 1999b).

The specification language CCSL is very closely related with the programming language CHARITY (Cockett and Fukushima, 1992; Schroeder, 1997). In CHARITY one programs only with initial algebras and final coalgebras. Thus, CCSL is the perfect specification language for CHARITY programs.

There are many different approaches that lead to a coalgebraic logic. Some authors are inspired by the idea that a logic for coalgebras (which are dualized algebras) should be based on dualized equations. Such *coequations* are presented in (Corradini, 1998; Cîrstea, 1999). In this work Corradini and Cîrstea present sound and complete deduction calculi for a restricted set of coalgebras. Binary methods do not fit into their notion of destructor signatures. The work in the present thesis is not so much concerned about complete deduction calculi. The primary goal here is to provide an expressive and convenient specification environment based on coalgebras.

[7] I use the term *coalgebraic logic* as a generic term for all logics that can (potentially) be used in coalgebraic specification. So the logic described in (Moss, 1999), which is called "coalgebraic logic" there, is one particular example of a coalgebraic logic.

Many papers analyse the connection between coalgebras and modal logics, for instance (Moss, 1999; Rößiger, 2000a; Kurz, 2000; Hughes, 2001). All these papers describe different modal logics for coalgebras. Moss describes characterising formulae (that is, formulae that describe a state up to observable equivalence). Rößiger describes a complete deduction calculus for his logics and uses modal logic to construct final coalgebras. Kurz and Huges employ modal logic in their work about a Birkhoff like theorem for coalgebras. The development of these modal logics has been driven by mathematical interest. As a result these logics are not very practical when it comes to expressing properties in coalgebraic specifications. Therefore all this cited work on modal logics plays only a secondary role for the present thesis. However, in the design of the modal operators of CCSL ideas have been drawn from (Rößiger, 2000a).

Hidden algebra (Roşu, 2000; Goguen and Malcolm, 2000) is a branch of (multi-sorted) algebraic specification in which some sorts of an algebraic signature are considered as hidden sorts on which no direct observation is possible. Hidden sorts are intended to capture the state space of automata and of classes. A severe restriction in hidden algebra is that a hidden signature contains only operations $S_1 \times \cdots \times S_n \longrightarrow S_0$, where all the S_i are sorts. So in hidden algebra one has neither structured argument types nor structured result types. Using coalgebras one can model partial operations easily with coalgebras of the form $\mathsf{Self} \longrightarrow \mathsf{Self} + \mathbf{1}$. In hidden algebra one has to use subsorting.

Another difference between hidden algebra and coalgebraic specification is the approach to define behavioural equivalence. In coalgebraic specification one uses bisimulations, a notion with which one can compare the behaviour of different models. Hidden algebra uses the approach of Reichel (Reichel, 1985) of visible contexts to define (what they call) hidden congruences. As soon as binary methods are present, hidden congruences can only compare states of one model. Bisimulations for coalgebras of polynomial functors form a complete lattice (Rutten, 2000). However, bisimulations for extended polynomial functors are not closed under union (Fact 3.3.7, but see also Proposition 3.4.30 and Theorem 3.5.9). In contrast, in hidden algebra one has always a greatest hidden congruence even in the presence of binary methods (Roşu, 2000).

Hennicker and Bidoit describe in (Hennicker and Bidoit, 1999) a logical framework for the specification of observable behaviour of systems. The approach described there is very similar to hidden algebra. Hennicker and Bidoit also use algebraic signatures with hidden sorts and exploit Reichels visible contexts to define their notion of observational equality. As in hidden algebra Hennicker and Bidoit do not allow structured argument or result types. Further their notion of observational equality cannot relate states of different models.

There are many other environments for software specification. (Kellomäki, 1997) describes the specification language DisCo for reactive systems. In DisCo one can describe the data fields of objects as state charts and specify actions in which the objects engage. The common framework initiative (Mosses, 1997) develops the Common Algebraic

Specification Language CASL[8] (Mossakowski, 2000). There are numerous sublanguages of CASL that correspond to the various logics that have been used in algebraic specification. However, there is no possibility to describe behavioural types in CASL.

The universal modelling language UML (Fowler, 1999; OMG, 2001) aims (as CCSL does) at the abstract description of object-oriented software systems. The UML is mainly a graphical language that is more concerned about the design process of software systems (and not so much about a formal description of the software). The Object Constraint Language OCL (Warmer and Kleppe, 1999; OMG, 1997) is currently developed as a logic for the UML. A comparison between CCSL and the official UML is impossible at the time of writing because the UML (and in particular OCL) does not have a precise semantics (yet). A comparison using a particular semantics of the UML that has been proposed in the past (for instance (Clark et al., 2001)) would be possible. However, such a comparison is beyond the scope of this thesis. A translation of a simple UML example (see Subsection 4.10.3 on page 238) suggests that a large class of UML class diagrams and OCL constraints can equivalently be expressed in CCSL. An embedding of CCSL into (the formal parts of) the UML fails because of the limited expressive power of OCL. Further the UML is very much focused on the object-oriented paradigm. Similar to many object-oriented programming languages it lacks support for algebraic data types.

[8] Not to be mixed up with the Custom Attack Simulation Language (CASL) (Vigna et al., 2000; Secure Networks, 1998).

2. Categorical Preliminaries

In this chapter I introduce the general notions, especially from category theory, that I use in the following chapters. The first section introduces the very basics: categories, functors, natural transformations, and adjunctions. The second section gives definitions for the bicartesian structure in a category to fix the notation for the following chapters. Section 2.3 defines basic notions from the theory of fibrations. I follow very closely (Jacobs, 1999a). With one exception, one could also say, that Section 2.3 is the simplified extract of (Jacobs, 1999a) that is necessary to follow the later chapters of this thesis. The exception is that I discuss the notion of cofibredness in more detail than Jacobs does. The fourth section explores two fibrations in more detail: The fibration of typed predicates and that of typed relations. These two fibrations are important in Chapter 3. The last two sections introduce algebras and coalgebras. In preparation of Chapter 3 the section on coalgebras describes the notions of bisimulation and invariant and presents many standard results from the literature.

No section of this chapter attempts to cover its subject completely. Rather, this chapter presents the material that is necessary for the following chapters. The first three sections about cartesian closed categories and fibrations present only standard definitions and results from the literature. Section 2.4 contains (besides many standard results) some new material (namely the results about cofibredness, see Example 2.4.16 and Lemma 2.4.17 on page 46ff).

The standard reference for category theory is (Mac Lane, 1997), but for computer scientists I would recommend (Barr and Wells, 1995). An introduction into the theory of fibrations is in (Jacobs, 1999a) and in (Phoa, 1992). Besides category theory I use basic set theory in this thesis, which I assume to be known. In the examples and in Chapter 4 I use notation from type theory. These things will be explained when they occur.

All the notation used and introduced in the following is standard, with one exception: The simple arrow ($\longrightarrow$). It is used with (on first sight) quite different interpretations in all of category theory, type theory, and logic. Therefore I depart from the standard notation and use the simple arrow solely to separate the domain and the codomain of a morphism (or of a function) as in $f : X \longrightarrow Y$. The exponent (i.e., the function space) is written as $X \Rightarrow Y$, the function type as $\sigma \Rightarrow \tau$. Implication is most times spelled out (i.e., *implies*). If not I use $F \supset G$ to denote that F implies G.

2.1. Categories, Functors and Adjunctions

Categories abstract from the set-theoretic element-wise thinking by considering only objects and *structure-preserving* mappings between them. A category $\mathbb{C}$ consists of a class of *objects* denoted by $|\mathbb{C}|$ and, for each pair $X, Y \in |\mathbb{C}|$ of objects, a class $\mathbb{C}(X, Y)$ of *morphisms* or *arrows* from X to Y. In category theory one prefers the notation $f : X \longrightarrow Y$ or $X \xrightarrow{f} Y$ instead of $f \in \mathbb{C}(X, Y)$ to denote that f is a morphism with *domain* X and *codomain* Y. The class $\mathbb{C}(X, Y)$ is often called the *homset* of X and Y. Two arrows f and g are *composable* if the codomain of f equals the domain of g. To form a category the classes of objects and morphisms must fulfil the following conditions. There must be an *associative composition* operation $- \circ -$ that assigns to each pair of composable morphisms $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ a morphism $g \circ f : X \longrightarrow Z$. For each object X, there must be an *identity morphism* $\mathrm{id}_X : X \longrightarrow X$ which is an identity for composition, that is $f \circ \mathrm{id}_X = \mathrm{id}_Y \circ f = f$.

A category $\mathbb{D}$ is a *subcategory* of $\mathbb{C}$ if $|\mathbb{D}| \subseteq |\mathbb{C}|$ and $\mathbb{D}(X, Y) \subseteq \mathbb{C}(X, Y)$ for all X, Y. Further, composition and identities in $\mathbb{D}$ must be as in $\mathbb{C}$. A category $\mathbb{C}$ is *locally small* if all classes $\mathbb{C}(X, Y)$ are proper sets. If, additionally, the class of objects is a set then $\mathbb{C}$ is *small.* As foundation I assume a nested hierarchy of collections as in (Adámek et al., 1990; Bénabou, 1985).

In correspondence with standard notation I use outlined letters like $\mathbb{C}$ to denote arbitrary categories, uppercase Latin letters like A, B, C for objects and lower case letters like f, g for morphisms.

Example 2.1.1

Category of Sets and total functions. The canonical example of a category is the category **Set** of sets and (total) functions. The objects of this category are the sets. The morphisms from a set M to a set N are all (total) functions $M \longrightarrow N$. The identities are the identity functions and composition of morphisms is given by composition of functions. The category **Set** is locally small but not small.

Provability in first-order logic. First-order logic over a signature Σ gives rise to the following category: objects are formulae F, G of first-order logic. Morphisms are entailments: There is a morphism $F \longrightarrow G$ if and only if $F \vdash G$ is provable in first order logic.

Note that in this category there is at most one morphism between any two objects. A category with this property is called a *preorder category.* ■

A morphism $f : X \longrightarrow Y$ in a category $\mathbb{C}$ is an *isomorphism* if there exists an inverse morphism $f^{-1} : Y \longrightarrow X$ such that $f^{-1} \circ f = \mathrm{id}_X$ and $f \circ f^{-1} = \mathrm{id}_Y$. I use $X \cong Y$ to denote that X and Y are isomorphic, that is, that there exists an isomorphism between them. In the category **Set** the isomorphisms are the bijective functions.

For any category $\mathbb{C}$ there is the *opposite category* $\mathbb{C}^{\mathrm{op}}$. It has the same objects as $\mathbb{C}$, but the morphisms are reversed: $\mathbb{C}^{\mathrm{op}}(X, Y) = \mathbb{C}(Y, X)$. So $f : X \longrightarrow Y$ is a morphism in $\mathbb{C}^{\mathrm{op}}$ if $f : Y \longrightarrow X$ is a morphism of $\mathbb{C}$.

Assume now an additional category $\mathbb{D}$. The *product category* $\mathbb{C} \times \mathbb{D}$ contains as objects pairs (X, Y), where X is an object of $\mathbb{C}$ and Y is an object of $\mathbb{D}$. The morphisms of $\mathbb{C} \times \mathbb{D}$ are also pairs $(f, g) : (U, V) \longrightarrow (X, Y)$ such that $f : U \longrightarrow X$ is a morphism of $\mathbb{C}$ and $g : V \longrightarrow Y$ is a morphism of $\mathbb{D}$. It is easy to check that $\mathbb{C} \times \mathbb{D}$ is indeed a category with composition defined pointwise.

In category theory one often draws *diagrams* like

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[dr, "h"'] & B \arrow[d, "g"] \\
 & C
\end{tikzcd}$$

to denote that f, g, and h are morphisms between the objects A, B, and C. Such a diagram *commutes* if, for any two objects in the diagram, the composition along any path between these two objects yields equal morphisms. So the above diagram commutes if $g \circ f = h$.

Consider two arrows $f : A \longrightarrow C$ and $g : B \longrightarrow C$ with common codomain. An object X together with two morphisms $u : X \longrightarrow A$ and $v : X \longrightarrow B$ is called a *pullback* (for f and g) and displayed as

$$\begin{tikzcd}
X \arrow[r, "u"] \arrow[d, "v"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "f"] \\
B \arrow[r, "g"] & C
\end{tikzcd}$$

if $f \circ u = g \circ v$ (the diagram commutes) and if additionally the following holds. For any object Y with morphisms $h : Y \longrightarrow A$ and $k : Y \longrightarrow B$ such that $f \circ h = g \circ k$ there exists a unique morphism $p : Y \longrightarrow X$ such that $u \circ p = h$ and $v \circ p = k$. Diagrammatically:

$$\begin{tikzcd}
Y \arrow[drr, bend left, "h"] \arrow[ddr, bend right, "k"'] \arrow[dr, dashed, "p"] & & \\
 & X \arrow[r, "u"] \arrow[d, "v"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "f"] \\
 & B \arrow[r, "g"] & C
\end{tikzcd}$$

If the mediating morphism p does exist but is not unique, the same structure is called a *weak pullback*.

In category theory one often works with combined uniqueness/existence properties like in the definition of pullbacks. Such properties are often used to prove that two morphisms are equal: If, in the above situation, I can construct a morphism $p' : Y \longrightarrow X$ such that $u \circ p' = h$ and $v \circ p' = k$, then by the pullback property I can conclude that

$p = p'$. Morphisms that are known to be unique are often drawn as dashed arrow $\dashrightarrow$ like the p in the preceding diagram.

A *functor* is a structure-preserving map between categories. Assume two categories $\mathbb{C}$ and $\mathbb{D}$. A mapping F that assigns objects and morphisms of $\mathbb{C}$ to, respectively, objects and morphism of $\mathbb{D}$ is a functor $\mathbb{C} \longrightarrow \mathbb{D}$ in case the following holds: A morphism $f : X \longrightarrow Y$ of $\mathbb{C}$ is mapped to a morphism $F(f) : F(X) \longrightarrow F(Y)$ in $\mathbb{D}$. Further, F must preserve identities, $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$, and composition $F(g \circ f) = F(g) \circ F(f)$. (Note that the standard notation makes use of overloading here, a functor $F : \mathbb{C} \longrightarrow \mathbb{D}$ is both a mapping $F : |\mathbb{C}| \longrightarrow |\mathbb{D}|$ and a mapping $F : \mathbb{C}(X, Y) \longrightarrow \mathbb{D}(F(X), F(Y))$ for all $X, Y \in |\mathbb{C}|$.) A functor $\mathbb{C} \longrightarrow \mathbb{C}$ with identical domain and codomain is called an *endofunctor*. The identity functor is written $\mathsf{Id}_{\mathbb{C}} : \mathbb{C} \longrightarrow \mathbb{C}$, the composition of two functors $F : \mathbb{C} \longrightarrow \mathbb{D}$ and $G : \mathbb{D} \longrightarrow \mathbb{E}$ is $GF : \mathbb{C} \longrightarrow \mathbb{E}$. For a functor $G : \mathbb{C} \times \mathbb{D} \longrightarrow \mathbb{E}$ I write $G(f, X)$ for $G(f, \mathrm{id}_X)$.

All small categories and the functors between them form the category **Cat**.

One can think of functors as describing (polymorphic) constructions on data (i.e., on objects). For instance, let $\mathsf{Tup} : \mathbf{Set} \longrightarrow \mathbf{Set}$ be the functor that is defined as $\mathsf{Tup}(X) \overset{\mathrm{def}}{=} X \times \mathbb{N}$, where $\mathbb{N}$ denotes the set of natural numbers. The functor Tup can be seen as the construction that forms the set of tuples $X \times \mathbb{N}$ for every parameter X. Type expressions in programming languages are typically modelled with functors. The idea carries on. One can think of functors as modelling the structural aspects of interfaces of software modules. In fact, almost every functor in this thesis can be seen as describing some interface. Under this interpretation a morphism corresponds to a particular operation of a software module or to a whole program.

Functors as just described preserve the direction of morphisms. To emphasise this fact, such functors are sometimes called *covariant functors*. Functors that invert the direction of morphisms are called *contravariant functors*. Formally, a contravariant functor $\mathbb{C} \longrightarrow \mathbb{D}$ is a covariant functor $\mathbb{C}^{\mathrm{op}} \longrightarrow \mathbb{D}$. We will see examples of contravariant functors below.

Functors are related by *natural transformations*. Assume two (covariant) functors $F, G : \mathbb{C} \longrightarrow \mathbb{D}$. A collection $\eta_X : F(X) \longrightarrow G(X)$ of morphisms in $\mathbb{D}$ indexed by the objects of $\mathbb{C}$ is a natural transformation $\eta : F \Longrightarrow G$, if for each morphism $f : X \longrightarrow Y$ of $\mathbb{C}$ the following diagram in $\mathbb{D}$ commutes:

$$\begin{tikzcd}
X \arrow[d, "f"] & F(X) \arrow[r, "\eta_X"] \arrow[d, "F(f)"'] & G(X) \arrow[d, "G(f)"] \\
Y & F(Y) \arrow[r, "\eta_Y"] & G(Y)
\end{tikzcd}$$

This diagram sits over

$$\begin{tikzcd}
\mathbb{C} \arrow[r, shift left, "F"] \arrow[r, shift right, "G"'] & \mathbb{D}
\end{tikzcd}$$

In the following I display categories (and functors) below the diagrams where this increases clarity.

A very important concept of category theory is that of an *adjunction*. An adjunction relates two categories by a pair of functors. Adjunctions are important because their existence imposes specific structure on the categories they relate. Further, they allow one to transfer results between the related categories. Proofs in category theory often proceed by moving forward and backward over an adjunction.

Consider two functors in opposite directions: a functor $F : \mathbb{C} \longrightarrow \mathbb{D}$ and a functor $G : \mathbb{D} \longrightarrow \mathbb{C}$. Both form an *adjunction*, written $F \dashv G : \mathbb{D} \longrightarrow \mathbb{C}$, if, for all $X \in |\mathbb{C}|$ and $Y \in |\mathbb{D}|$, there exists a bijection between the homsets $\mathbb{D}(F(X), Y)$ and $\mathbb{C}(X, G(Y))$ fulfilling the *naturality condition*. I denote this bijection with $-^{\wedge} : \mathbb{C}(X, G(Y)) \longrightarrow \mathbb{D}(F(X), Y)$. So if $f : X \longrightarrow G(Y)$ is a morphism in $\mathbb{C}$, then the adjunction $F \dashv G$ determines a unique morphism $f^{\wedge} : F(X) \longrightarrow Y$ in $\mathbb{D}$. The inverse of $-^{\wedge}$ is $-^{\vee}$: If $g : F(X) \longrightarrow Y$ is a morphism in $\mathbb{D}$, then the adjunction gives a unique morphism $g^{\vee} : X \longrightarrow G(Y)$ in $\mathbb{C}$. The *naturality condition* says that the bijection between the homsets commutes with composition from the left and the right. For the first case (composition from the left) consider the following situation:

$$\begin{tikzcd}
W \arrow[d, "h"'] \arrow[dr, "f \circ h"] & \\
X \arrow[r, "f"] & G(Y)
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
F(W) \arrow[d, "F(h)"'] \arrow[dr, "(f \circ h)^{\wedge}"] & \\
F(X) \arrow[r, "f^{\wedge}"] & Y
\end{tikzcd}$$

$$\begin{tikzcd}
\mathbb{C} \arrow[rr, bend left, "F"] & \perp & \mathbb{D} \arrow[ll, bend left, "G"]
\end{tikzcd}$$

In $\mathbb{C}$ the morphism h can be composed with f. Naturality now means, that for arbitrary f and h the right triangle in $\mathbb{D}$ commutes, that is, that $f^{\wedge} \circ F(h) = (f \circ h)^{\wedge}$. The second case (composition from the right) is very similar: If we have two morphisms $F(X) \xrightarrow{g} Y \xrightarrow{k} Z$ in $\mathbb{D}$ then it must hold that $G(k) \circ g^{\vee} = (k \circ g)^{\vee}$ in $\mathbb{C}$.

The morphisms $f^{\wedge}$ and $g^{\vee}$ are often called the *adjoint transposes* of f and g, respectively. If F and G form a pair of adjoint functors $F \dashv G$ then F is the *left adjoint* of G and, vice versa, G is the *right adjoint* of F. Every adjunction involves two natural transformations. The *unit*, usually denoted by $\eta : \mathsf{Id}_{\mathbb{C}} \Longrightarrow GF$ and the *counit*, denoted by $\varepsilon : FG \Longrightarrow \mathsf{Id}_{\mathbb{D}}$. Both natural transformation arise as adjoint transposes of identity morphisms: $\eta_X = \mathrm{id}_{F(X)}{}^{\vee}$ and $\varepsilon_Y = \mathrm{id}_{G(Y)}{}^{\wedge}$.

There exist a number of equivalent definitions for adjunction, compare Theorem IV.2. in (Mac Lane, 1997) or Section 13.3 in (Barr and Wells, 1995).

2.2. Bicartesian Closed Categories

Bicartesian closed categories are categories with certain additional structure. Bicartesian closed categories are in the centre of interest of this thesis, because they form a suitable abstraction of the type theoretic settings of most programming languages. This subsection introduces the categorical structure that is necessary to model tuple or record types (*products*), union or variant types (*coproducts*), and function types (*exponents*).

An object $\mathbf{0}$ in a category $\mathbb{C}$ is called *initial* if, for each object X of $\mathbb{C}$, there exists exactly one morphism $\mathbf{0} \longrightarrow X$. Similarly, an object $\mathbf{1} \in |\mathbb{C}|$ is called *final* if, for all objects X of $\mathbb{C}$, there exists exactly one morphism $X \longrightarrow \mathbf{1}$ (some authors prefer the term *terminal object*). The morphism into the final object is denoted by $!_X$. Note that $\mathbb{C}$ has a final object just in case $\mathbb{C}^{\mathrm{op}}$ has an initial one.

Consider the category $\mathbb{1}$ that consists of one object and only one arrow, the identity morphism for that object. For any category $\mathbb{C}$ there is a functor $\mathbf{!} : \mathbb{C} \longrightarrow \mathbb{1}$ that sends all objects and all morphisms of $\mathbb{C}$ to the only object and the only morphism of $\mathbb{1}$, respectively (turning $\mathbb{1}$ into the final object in **Cat**). The category $\mathbb{C}$ has an initial object if and only if this functor $\mathbf{!}$ has a left adjoint. The left adjoint maps the only object of $\mathbb{1}$ to the initial object of $\mathbb{C}$. The counit of this adjunction gives the unique morphisms out of the initial object. Similarly, if $\mathbb{C}$ has a final object, then $\mathbf{!}$ has a right adjoint that picks out the final object. The unit of this latter adjunction gives the unique morphisms into the final object.

The category $\mathbb{C}$ has *finite products* if it has a final object (the empty product) and if for all pairs X and Y of objects of $\mathbb{C}$ there exists an object $X \times Y$ in $\mathbb{C}$ with the following properties: There exist two *projection morphisms* $X \xleftarrow{\pi_1} X \times Y \xrightarrow{\pi_2} Y$. And for any object Z of $\mathbb{C}$ with a pair of morphisms $X \xleftarrow{f} Z \xrightarrow{g} Y$, there exists exactly one morphism $Z \longrightarrow X \times Y$, denoted by $\langle f, g \rangle$, such that the diagram below commutes.

$$\begin{tikzcd}
& Z \arrow[dl, "f"'] \arrow[d, dashed, "\langle f,g \rangle"] \arrow[dr, "g"] & \\
X & X \times Y \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & Y
\end{tikzcd}$$

Consider the functor $\Delta : \mathbb{C} \longrightarrow \mathbb{C} \times \mathbb{C}$ that sends objects X and morphisms f to the pairs (X, X) and (f, f), respectively. Assuming an Axiom of Choice of sufficient strength, one can show that the category $\mathbb{C}$ has finite products if and only if there is an adjunction $\Delta \dashv \times : \mathbb{C} \times \mathbb{C} \longrightarrow \mathbb{C}$. The right adjoint $\times$ sends pairs of objects to their product and pairs of morphisms f and g to $f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle$.

Coproducts are the duals of products. This means that $\mathbb{C}$ has coproducts if $\mathbb{C}^{\mathrm{op}}$ has products or, more informally, coproducts are products with all morphisms reversed: The category $\mathbb{C}$ has *finite coproducts* if it has an initial object (the empty coproduct) and if for all pairs X and Y of objects of $\mathbb{C}$ there exists an object $X + Y$ in $\mathbb{C}$ with the following properties: There exist two *injection morphisms* $X \xrightarrow{\kappa_1} X + Y \xleftarrow{\kappa_2} Y$. And for any object Z of $\mathbb{C}$ with a pair of morphisms $X \xrightarrow{f} Z \xleftarrow{g} Y$ there exists exactly one morphism $X + Y \longrightarrow Z$, denoted by $[f, g]$ such that the diagram below commutes.

$$\begin{tikzcd}
X \arrow[r, "\kappa_1"] \arrow[dr, "f"'] & X + Y \arrow[d, dashed, "{[f,g]}"] & Y \arrow[l, "\kappa_2"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd} \tag{2.1}$$

Similar to products, finite coproducts give rise to a left adjoint to Δ. Its action on morphisms is given by $f + g = [\kappa_1 \circ f, \kappa_2 \circ g]$.

A category $\mathbb{C}$ with products has *exponents* if for each pair of objects X and Y of $\mathbb{C}$ there exists an object $X \Rightarrow Y$ and a morphism $\mathsf{eval}_{X,Y} : (X \Rightarrow Y) \times X \longrightarrow Y$ with the following property: For each object Z with a morphism $f : Z \times X \longrightarrow Y$ there exists exactly one morphism $\lambda f : Z \longrightarrow X \Rightarrow Y$ such that the following diagram commutes.

$$\begin{tikzcd}
X \Rightarrow Y & & (X \Rightarrow Y) \times X \arrow[r, "\mathsf{eval}"] & Y \\
Z \arrow[u, dashed, "\lambda f"] & & Z \times X \arrow[u, "\lambda f \times \mathrm{id}_X"] \arrow[ur, "f"'] &
\end{tikzcd}$$

Exponents can also be characterised by an adjunction. For a fixed object X of $\mathbb{C}$ the functor $- \times X$ is left adjoint to $X \Rightarrow -$. The functor $X \Rightarrow -$ can be extended to a mixed variance functor $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$. This latter functor maps two morphisms $f : V \longrightarrow U$ and $g : X \longrightarrow Y$ to $f \Rightarrow g : U \Rightarrow X \longrightarrow V \Rightarrow Y = \lambda\big(g \circ \mathsf{eval}_{U,X} \circ (\mathrm{id} \times f)\big)$. The following diagram illustrates this construction.

$$\begin{tikzcd}
(V \Rightarrow Y) \times V \arrow[rr, "\mathsf{eval}"] & & Y \\
(U \Rightarrow X) \times V \arrow[u, "(f \Rightarrow g) \times \mathrm{id}_V"] \arrow[r, "\mathrm{id}_{U \Rightarrow X} \times f"] & (U \Rightarrow X) \times U \arrow[r, "\mathsf{eval}"] & X \arrow[u, "g"']
\end{tikzcd}$$

A category with finite products and exponents is called *cartesian closed*. If additionally it has finite coproducts, then it is *bicartesian closed*.

Example 2.2.1 (Continuing Example 2.1.1.) The initial object in **Set** is the empty set with the empty function as unique morphism. Every one-element set is a final object in **Set**. The product is given by the cartesian product of sets: $M \times N = \{(m, n) \mid m \in M, n \in N\}$. For two functions $f : S \longrightarrow M$ and $g : S \longrightarrow N$ their pairing $\langle f, g \rangle$ is given by $\lambda s : S \,.\, (f(s), g(s))$.

The last expression involves notation from typed lambda calculus (Barendregt, 1992). There, $\lambda x : X \,.\, f(x)$ denotes a function that maps elements x of type X to $f(x)$. If M is a term of functional type and if N is a term of its domain type, then function application is written as MN or, to avoid ambiguities, as $M(N)$ or even $(M)(N)$. If M is a λ-expression, like in $(\lambda n : \mathbb{N} \,.\, n + 1)\,(5)$ then this is called a redex, which can be simplified by substituting the argument 5 for the formal parameter n. In the example the reduction would result with the term $5 + 1$.

The disjoint union of two sets $M \uplus N = (M \times \{1\}) \cup (N \times \{2\})$ forms the coproduct in **Set**. The injection functions pair their arguments with 1 or 2. For instance $\kappa_1 = \lambda m : M \,.\, (m, 1)$. The copairing of two functions is given by case distinction (or pattern matching):

$$[f, g] \quad = \quad \lambda z : M \uplus N \,.\, \mathsf{cases}\ z\ \mathsf{of}\ \kappa_1\, m : f(m),\quad \kappa_2\, n : g(n)\ \mathsf{endcases}$$

The set of all functions $M \longrightarrow N$ is the exponent of M and N. The canonical map $\mathsf{eval}_{M,N}$ is function evaluation $\lambda f : M \Rightarrow N,\ m : M \,.\, f(m)$. And for a function $f : Z \times X \longrightarrow Y$ the unique function λf is given by $\lambda z : Z \,.\, (\lambda x : X \,.\, f(z, x))$. ■

Example 2.2.2 (Type Theory and Classifying category) A type theory is a formal calculus about terms and types, see (Jacobs, 1999a) for a comprehensive study. Simple type theories gives rise to syntactically constructed categories, the *classifying categories*. The word 'simple' here refers to the fact that types do not contain type variables in these type theories (so there is no polymorphism in a simple type theory). The classifying category is denoted with $\mathcal{C}\!\ell$. It can be used to study the given type theory, for instance its semantics. Here I sketch a simple type theory with products and exponent types (called $\lambda 1_\times$ in (Jacobs, 1999a)) and its classifying category. This example paves the way for Example 2.3.4 (on page 27) and also for the first sections of Chapter 4 that describe the polymorphic type theory of the specification language CCSL.

Arbitrary types are usually denoted with lowercase Greek variables like τ, σ. In a type theory with products one always has the unit type $\mathbf{1}$ (for the empty product). More atomic types are usually drawn from a signature, but this is not important for this example. The set of types is inductively defined: For any two types τ and σ their product $\tau \times \sigma$ and their exponent[1] $\tau \Rightarrow \sigma$ is a type. Sometimes one has a derivation system that allows one to derive *typing judgements* $\vdash \tau : \mathsf{Type}$ precisely if τ is a valid type in the type theory. For this example such a derivation system would contain (among others) the following rules:

Unit $\qquad$ **Product** $\qquad$ **Exponent**

$$\frac{}{\vdash \mathbf{1} : \mathsf{Type}} \qquad \frac{\vdash \tau : \mathsf{Type} \quad \vdash \sigma : \mathsf{Type}}{\vdash \tau \times \sigma : \mathsf{Type}} \qquad \frac{\vdash \tau : \mathsf{Type} \quad \vdash \sigma : \mathsf{Type}}{\vdash \tau \Rightarrow \sigma : \mathsf{Type}}$$

In these rules the assumptions are above the line (so the rule Unit has no assumptions) and the conclusion is below. For instance the rule Product should be read as follows:

[1] In the literature the exponent type is usually written as $\tau \longrightarrow \sigma$, but in this thesis the simple arrow $\longrightarrow$ is reserved for morphisms.

If both τ and σ are types (i.e., there are derivations for the sequents $\vdash \tau : \mathsf{Type}$ and $\vdash \sigma : \mathsf{Type}$) then also $\tau \times \sigma$ is a type (i.e., you can build a derivation for $\vdash \tau \times \sigma : \mathsf{Type}$ by plugging the two derivations for τ and σ into the assumptions of the product rule).

In addition to the set of types, the type theory describes how to build *terms* and which type a given term has. Arbitrary terms are denoted with lowercase Latin letters like s, t. Terms may contain variables. For variables I use also lowercase Latin letters like x, y. A variable is free if it is not bound by a variable binder like λ or $\forall$. A term with no free variables is a closed term. In general the type of the free variables is important, therefore the free variables that may occur in a term t are collected in a *context* $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. Formally a context is a finite list of variable declarations $x_i : \tau_i$, where all the variables are pairwise distinct.[2] A *term judgement* has the form

$$\Gamma \quad \vdash \quad t : \sigma$$

where Γ is a context, t is a term, and σ is a type. A term judgement is the formal statement that the term t has type σ if the free variables in t have types according to Γ. The main ingredient of a type theory is a derivation system for term judgements. If a judgement $\Gamma \vdash t : \sigma$ can be derived one says that the term t *inhabits* the type σ. A type is empty if it has no inhabitants. The derivation system ensures that a judgement $\Gamma \vdash t : \sigma$ can only be derived if all free variables of t are declared in Γ.

A type theory with product and exponent types typically contains the following derivation rules:

ground terms

$$\frac{\vdash \tau_i : \mathsf{Type}}{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i} \qquad\qquad \frac{}{\Gamma \vdash * : \mathbf{1}}$$

product

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t : \tau}{\Gamma \vdash (s, t) : \sigma \times \tau} \qquad \frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \pi_1 t : \sigma} \qquad \frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash \pi_2 t : \tau}$$

exponent

$$\frac{\vdash \sigma : \mathsf{Type} \quad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x : \sigma . t \ : \ \sigma \Rightarrow \tau}\ x \notin \Gamma \qquad\qquad \frac{\Gamma \vdash t : \sigma \Rightarrow \tau \quad \Gamma \vdash s : \sigma}{\Gamma \vdash t\, s : \tau}$$

Note that in the derivation rule for lambda abstraction the side condition $x \notin \Gamma$ and the first assumption $\vdash \sigma : \mathsf{Type}$ are redundant: If $\Gamma, x : \sigma$ is a valid context then σ is

[2] This distinction can formally be ensured by using natural numbers as variables. See (Jacobs, 1999a) Section 2.1. Such rather technical issues are not relevant here.

a type and x cannot occur in Γ (because all variables in a context must be distinct). I prefer to state redundant assumptions if this makes important details explicit.

In the term $\lambda x : \sigma \,.\, t \;:\; \sigma \Rightarrow \tau$ the variable x is called a *bound variable.* It is clear that renaming a bound variable in a term t does not change the meaning of t. The process of renaming a bound variable is called *α–conversion* and two terms that differ only in the names of bound variables are called *α–equivalent.* One can avoid α–conversion if one represents bound variables with de Brujin indices (de Bruijn, 1972). Unfortunately, de Bruijn indices are not really suitable for human consumption.

Substitution denotes the process of replacing a free variable x in a term t with a term s. In this thesis I write $t[s/x]$ to denote the result of substituting the term s for the variable x in t. Substitution can be defined by induction on the term structure:

$$\begin{array}{lcll}
*[s/x] & = & * & \\
x[s/x] & = & s & \\
y[s/x] & = & y & \text{for } x \neq y \\
(t_1, t_2)[s/x] & = & (t_1[s/x], t_2[s/x]) & \\
(\pi_i\, t)[s/x] & = & \pi_i\,(t[s/x]) & \\
(\lambda x : \sigma \,.\, t)[s/x] & = & (\lambda x : \sigma \,.\, t) & \\
(\lambda y : \sigma \,.\, t)[s/x] & = & (\lambda y : \sigma \,.\, t[s/x]) & \text{if } x \neq y \text{ and } y \text{ is not free in } s \\
(t_1\, t_2)[s/x] & = & (t_1[s/x])\,(t_2[s/x]) &
\end{array}$$

The side condition for the substitution $(\lambda y : \sigma \,.\, t)[s/x]$ does not pose any problems, because one can always apply α–conversion and rename the bound variable y in t.

By induction over the structure of a derivation of a term judgement one can prove the rule

term substitution

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash t[s/x] : \tau}\; x \notin \Gamma$$

Another important ingredient of a type theory are the *conversions.* Conversions describe which terms are thought to be equal although their syntactic appearance is different. We already saw α–conversion for renaming bound variables. β–conversion deals with the intended operational behaviour of terms:

$$\begin{array}{rcl}
\pi_i(s_1, s_2) & = & s_i \\
(\lambda x : \sigma \,.\, t)\, s & = & t[s/x]
\end{array}$$

Extensionality of functions is expressed by η–conversion:

$$\lambda x : \sigma \,.\, s\, x \quad = \quad s$$

provided that x is not free in s. One usually extends conversion to an equivalence relation that is compatible with the term forming operations. Then we have also

$$\begin{array}{rcl} s \;=\; s' & \text{implies} & \lambda x:\sigma\,.\,s \;=\; \lambda x:\sigma\,.\,s' \\ s \;=\; s' \quad\wedge\quad t\;=t' & \text{implies} & (s,t) \;=\; (s',t') \end{array}$$

An important lemma, which one could prove now, is that conversion preserves types. That is, if $\Gamma \vdash t : \sigma$ and $t = s$ then also $\Gamma \vdash s : \sigma$. In the following, equivalence classes of terms under conversion are important. They are written as $[t] = \{s \mid s = t\}$.

For a simple type theory as the one sketched here the classifying category $\mathcal{C}\!\ell$ is defined as follows: Objects of $\mathcal{C}\!\ell$ are contexts $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$. For two contexts Γ and $\Delta = y_1 : \sigma_1, \ldots, y_m : \sigma_m$, a morphism $\Gamma \longrightarrow \Delta$ is an m–tuple $([t_1], \ldots, [t_m])$ of equivalence classes of terms such that $\Gamma \vdash t_i : \sigma_i$ can be inferred in the given type theory. The identity morphisms of $\mathcal{C}\!\ell$ are (equivalence classes of) variable tuples. For instance $([y_1], \ldots, [y_m]) : \Delta \longrightarrow \Delta$ is the identity morphism of the context Δ. Composition of morphisms is given by substitution. Assume two morphisms $([t_1], \ldots, [t_m]) : \Gamma \longrightarrow \Delta$ and $([s_1], \ldots, [s_k]) : \Delta \longrightarrow \Theta$. Their composition is then

$$([s_1[t_1/y_1, \ldots, t_m/y_m]\,], \ldots, [s_k[t_1/y_1, \ldots, t_m/y_m]\,]) \quad : \quad \Gamma \longrightarrow \Theta$$

The classifying category has finite products, even if the original type theory does not have product types. The product of two contexts is their concatenation (modulo a suitable renaming to avoid name clashes). The projection morphisms are the obvious tuples of variables. The empty context is a final object in $\mathcal{C}\!\ell$. ■

2.3. Fibrations

Fibrations provide a categorical formalisation of predicate logic. In Chapter 3 I investigate properties of invariants and bisimulations for an extended notion of coalgebra. Much of this reasoning will take place in the fibration of (typed) predicates. This section introduces the notions from fibred category theory that will be needed there.

Consider a functor $p : \mathbb{E} \longrightarrow \mathbb{B}$. If p is intended to be a fibration it is written in the following way: $\begin{smallmatrix}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{smallmatrix}$. Here, the category $\mathbb{B}$ is called the *base* and $\mathbb{E}$ is called the *total category*. An object X from the total category $\mathbb{E}$ is *above* or *over* an object I from the base $\mathbb{B}$ if $p\,X = I$. Similarly for morphisms: $f : X \longrightarrow Y$ is said to be above (or over) $u : I \longrightarrow J$ if $p\,f = u$ (which implies that X is over I and Y over J).

A morphism f in the total category $\mathbb{E}$ is *vertical* if it is over some identity, that is if $p\,f = \mathrm{id}_I$ for some object I. For every object I of the base there is the subcategory $\mathbb{E}_I$ of the total category that consists of all objects over I and all vertical morphisms between them. This subcategory $\mathbb{E}_I$ is called the *fibre over I*. If X and X' are objects over I, then $\mathbb{E}_I(X, X')$ denotes the homset in $\mathbb{E}_I$, that is, $\mathbb{E}_I(X, X')$ stands for all vertical morphisms from X to X'. For a morphism $u : I \longrightarrow J$ in the base and two objects X and Y above (respectively) I and J the set of morphisms over u is denoted by $\mathbb{E}_u(X, Y)$. Formally

$$\mathbb{E}_u(X,Y) \quad = \quad \{f \mid f \in \mathbb{E}(X,Y) \ \text{ and } \ p\,f = u\}$$

Definition 2.3.1 (Fibration) Consider a functor $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$.

1. A morphism $f : X \longrightarrow Y$ above $u : I \longrightarrow J$ is *cartesian over u* if for every morphism $g : Z \longrightarrow Y$, for which there exists a morphism $w : p\,Z \longrightarrow I$ such that $p\,g = u \circ w$, there exists a unique morphism $h : Z \longrightarrow X$ above w such that $g = f \circ h$.

$$\begin{tikzcd}
 & Z \arrow[dr, dashed, "h"'] \arrow[drr, bend left, "g"] & & \\
\mathbb{E} \arrow[dd, "p"'] & & X \arrow[r, "f"'] & Y \\
 & p\,Z \arrow[dr, "w"'] \arrow[drr, bend left, "p\,g"] & & \\
\mathbb{B} & & I \arrow[r, "u"'] & J
\end{tikzcd}$$

2. The functor $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a *fibration* if, for all objects Y in the total category and all morphisms $u : I \longrightarrow p\,Y$, there exists a cartesian morphism $f : X \longrightarrow Y$ over u.

If $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a fibration one also says that $\mathbb{E}$ is *fibred (via p) over* $\mathbb{B}$. Cartesian morphisms with the same codomain are unique up to isomorphism: If $f : X \longrightarrow Y$ and $f' : X' \longrightarrow Y$ are both cartesian morphisms over the same arrow in the base category then there exists a canonical (vertical) isomorphism $\varphi : X \longrightarrow X'$. This isomorphism is canonical in the sense that it commutes with f and f', that is, we have $f' \circ \varphi = f$ (which implies $f \circ \varphi^{-1} = f'$).

Because cartesian morphisms are unique up to isomorphism it makes sense to choose a particular cartesian morphism for every arrow in the base category. A fibration is *cloven* if it comes equipped with a choice of cartesian liftings for the morphisms of the base category. In this case I write $\widehat{u} : u^*\,Y \longrightarrow Y$ for the cartesian morphism over $u : I \longrightarrow p\,Y$. In a cloven fibration, for every morphism $u : I \longrightarrow J$, the operation u^* extends to a functor $\mathbb{E}_J \longrightarrow \mathbb{E}_I$ between the fibres. This functor u^* is called the *substitution functor along u*.

Assume a cloven fibration $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ and consider two composable morphisms in the base category: $I \xrightarrow{u} J \xrightarrow{v} K$. For any object Y above K there are the following two canonical isomorphisms: $u^*\,(v^*\,Y) \cong (v \circ u)^*\,Y$ and $\mathrm{id}_K^*\,Y \cong Y$. A fibration for which both isomorphisms are identities is called a *split fibration*. For the rest of this thesis I restrict my attention to cloven fibrations. (Every fibration can be turned into a cloven fibration by exploiting an Axiom of Choice of suitable strength.)

In a cloven fibration every morphism $f : X \longrightarrow Y$ in the total category can be decomposed into $f = \widehat{p\,f} \circ \widetilde{f}$, where $\widehat{p\,f}$ is the cartesian arrow over $p\,f$ and $\widetilde{f}$ is the vertical morphism that is uniquely determined by the properties of a fibration. Thus, we get for every $u : I \longrightarrow J$ in the base an isomorphism between homsets in the total category $\mathbb{E}_u(X, Y) \;\cong\; \mathbb{E}_I(X, u^*\,Y)$.

A *preorder fibration* is a fibration in which all the fibres are preorder categories, that is, there is at most on vertical morphisms between any two objects in any fibre. This section applies to fibrations in general (i.e., not only to preorder fibrations), but all concrete fibrations studied in the present thesis are preorder fibrations.

The contravariant nature of the exponent forces me to consider also cofibrations in this thesis (they are called *opfibrations* in (Jacobs, 1999a), Section 9.1). A functor $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a cofibration if it is a fibration when considered as a functor $\begin{array}{c}\mathbb{E}^{\mathrm{op}}\\ \downarrow p\\ \mathbb{B}^{\mathrm{op}}\end{array}$. For convenience I spell out the definition.

Definition 2.3.2 (Cofibration) Consider a functor $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$.

1. A morphism $f : X \longrightarrow Y$ above $u : I \longrightarrow J$ is *cocartesian over* u if for every $g : X \longrightarrow Z$, for which there exists a morphism $w : J \longrightarrow p\,Z$ such that $p\,g = w \circ u$, there exists a unique morphism $h : Y \longrightarrow Z$ above w such that $g = h \circ f$.

$$\begin{tikzcd}
 & & & Z \\
\mathbb{E} \arrow[dd, "p"'] & X \arrow[r, "f"'] \arrow[urr, bend left, "g"] & Y \arrow[ur, dashed, "h"'] & \\
 & & & p\,Z \\
\mathbb{B} & I \arrow[r, "u"'] \arrow[urr, bend left, "p\,g"] & J \arrow[ur, "w"'] &
\end{tikzcd}$$

2. The functor $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a *cofibration* if, for all objects X in the total category and all morphisms $u : p\,X \longrightarrow J$, there exists a cocartesian morphism $f : X \longrightarrow Y$ over u.

3. The functor $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a *bifibration* if it is a fibration and a cofibration at the same time.

The notions of a *cloven* or *split cofibration* are defined in the same way as for fibrations. A cofibration $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is cloven (split) if $\begin{array}{c}\mathbb{E}^{\mathrm{op}}\\ \downarrow p\\ \mathbb{B}^{\mathrm{op}}\end{array}$ is a cloven (split) fibration. From the structure of a cloven cofibration we obtain for every morphism $u : I \longrightarrow J$ in the base a *cosubstitution* or better *coproduct* functor $\mathbb{E}_I \longrightarrow \mathbb{E}_J$. This coproduct functor is denoted by $\coprod_u$. Similar to fibrations one obtains $\coprod_v (\coprod_u X) \cong \coprod_{v \circ u} X$ for any pair of composable arrows $I \xrightarrow{u} J \xrightarrow{v} K$ and $\coprod_{\mathrm{id}} X \cong X$.

Consider again a fibration $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$. The substitution functors u^* can have left and right adjoints. The left adjoint to u^* is called the *coproduct along* u and denoted by $\coprod_u$. The right adjoint $\prod_u \vdash u^*$ is called the *product along* u. The clash in naming and notation of the coproduct functor is intended and is justified in the following lemma.

Lemma 2.3.3 *A fibration* $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ *is also a cofibration if and only if each substitution functor* u^* *has a left adjoint* $\coprod_u$.

For illustration I copy the proof from (Jacobs, 1999a).

Proof Because $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a fibration we have an isomorphism between homsets for an arbitrary morphism $u : I \longrightarrow J$ in the base:

$$\mathbb{E}_I(X, u^* Y) \quad \cong \quad \mathbb{E}_u(X, Y) \tag{$*$}$$

An adjunction $\coprod_u \dashv u^*$ amounts to isomorphic homsets

$$\mathbb{E}_J(\textstyle\coprod_u X, Y) \quad \cong \quad \mathbb{E}_I(X, u^* Y) \tag{$\dagger$}$$

Combining both $(*)$ and $(\dagger)$ gives

$$\mathbb{E}_J(\textstyle\coprod_u X, Y) \quad \cong \quad \mathbb{E}_u(X, Y) \tag{$\ddagger$}$$

that is, that $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$ is a cofibration. Similarly $(\dagger)$ is obtained from $(*)$ and $(\ddagger)$. □

When working with a fibration one can distinguish different views. One can investigate the structure in the single fibres, the structure in the total category, and the structure between the fibres. An example of the structure between the fibres are the substitution functors and their adjoints. It often happens that a fibration possesses a given structure or property both in the total category and in all the fibre categories. It is therefore important to be precise and to distinguish the structure of the total category and the structure of the fibre.

A fibration has *fibred products* (*fibred coproducts* or *fibred exponents*) if each fibre category has products (coproducts or exponents, respectively) and if the substitution functors preserve this structure. That is, if $X \wedge Y$ is the product of two objects in the fibre over J and if $u : I \longrightarrow J$ is a morphism in the base, then it is required that $u^* (X \wedge Y) \cong (u^* X) \wedge (u^* Y)$. Because the main motivation for fibrations is logic, I use $\wedge$, $\vee$, and $\supset$ for the fibred product, the fibred coproduct, and the fibred exponent, respectively.

Example 2.3.4 In Section 2.4 (starting on page 34) I construct the fibration of typed predicates and that of typed relations and discuss their properties in detail. Here I show a syntactic example of a fibration: Consider a classifying category $\mathcal{C}\ell$ for a simple type theory with product types and exponents as presented in Example 2.2.2. A logic for this type theory forms a fibration over $\mathcal{C}\ell$ with additional structure depending on the properties of the logic. In this example I describe this fibration for predicate logic.

Predicate logic has rules for building *well-formed formulae* and for deriving valid sequents. Judgements for well-formed formulae have the form $\Gamma \vdash \varphi : \mathsf{Prop}$. If one views Prop as the special type of formulae, then this is just a term judgement for the type of formulae. Assuming that basic propositions are given as a set of judgements $x : \tau \vdash P(x) : \mathsf{Prop}$, there are the following rules for atomic formulae:

equality

$$\frac{\Gamma \vdash s : \tau \quad \Gamma \vdash t : \tau}{\Gamma \vdash s = t : \mathsf{Prop}}$$

predicate

$$\frac{\Gamma \vdash t : \tau \quad x : \tau \vdash P(x) : \mathsf{Prop}}{\Gamma \vdash P(t) : \mathsf{Prop}}$$

Rules for complex formulae are for instance:

conjunction

$$\frac{\Gamma \vdash \varphi : \mathsf{Prop} \quad \Gamma \vdash \psi : \mathsf{Prop}}{\Gamma \vdash \varphi \wedge \psi : \mathsf{Prop}}$$

disjunction

$$\frac{\Gamma \vdash \varphi : \mathsf{Prop} \quad \Gamma \vdash \psi : \mathsf{Prop}}{\Gamma \vdash \varphi \vee \psi : \mathsf{Prop}}$$

universal quantification

$$\frac{\vdash \tau : \mathsf{Type} \quad \Gamma, x : \tau \vdash \varphi : \mathsf{Prop}}{\Gamma \vdash \forall x : \tau \,.\, \varphi : \mathsf{Prop}} \; x \notin \Gamma$$

existential quantification

$$\frac{\vdash \tau : \mathsf{Type} \quad \Gamma, x : \tau \vdash \varphi : \mathsf{Prop}}{\Gamma \vdash \exists x : \tau \,.\, \varphi : \mathsf{Prop}} \; x \notin \Gamma$$

When discussing a logic attention is often only devoted to the derivation system for valid logical entailments. Judgements for logical entailments have the form $\Gamma \mid \varphi \vdash \psi$. Here Γ is again a context of variable declarations, and φ and ψ are well-formed propositions under the context Γ. The intended meaning of $\Gamma \mid \varphi \vdash \psi$ is that ψ can be deduced (by using the rules of the logic) from φ. Rules for deriving valid entailments are for instance:

and

$$\frac{\Gamma \mid \varphi \vdash \psi \quad \Gamma \mid \varphi \vdash \rho}{\Gamma \mid \varphi \vdash \psi \wedge \rho}$$

axiom

$$\frac{}{\Gamma \mid \varphi \vdash \varphi}$$

cut

$$\frac{\Gamma \mid \varphi \vdash \psi \quad \Gamma \mid \psi \vdash \rho}{\Gamma \mid \varphi \vdash \rho}$$

substitution

$$\frac{\Gamma, x : \tau \mid \varphi \vdash \psi \quad \Gamma \vdash t : \tau}{\Gamma \mid \varphi[t/x] \vdash \psi[t/x]}$$

The substitution rule is usually a derived rule.

Similar to the classifying category there exists a category $\mathcal{L}$ that describes this logic and the valid implications in it. Recall from Example 2.2.2 that the classifying category has variable contexts Γ as objects and tuples of equivalence classes of terms $([t_1], \dots, [t_n])$ as morphisms. In the following I identify an equivalence class of terms with its representative, writing t instead of $[t]$. Objects of $\mathcal{L}$ are well-formed propositions $\Gamma \vdash \varphi : \mathsf{Prop}$. Assume now variable contexts $\Gamma = x_1 : \tau_1, \dots, x_m : \tau_m$ and $\Delta = y_1 : \sigma_1, \dots, y_n : \sigma_n$ and two propositions $\Gamma \vdash \varphi : \mathsf{Prop}$ and $\Delta \vdash \psi : \mathsf{Prop}$. A morphism $(\Gamma \vdash \varphi) \longrightarrow (\Delta \vdash \psi)$ in $\mathcal{L}$ is a morphism $(t_1, \dots, t_n) : \Gamma \longrightarrow \Delta$ in $\mathcal{C}\ell$ such that one can formally infer $\Gamma \mid \varphi \vdash \psi[t_1/y_1, \dots, t_n/y_n]$.

The $\mathcal{L}$ that I just described is indeed a category. It has identity morphisms, because of the axiom rule. The composition in $\mathcal{L}$ is given by the composition in $\mathcal{C}\ell$. Consider for instance the propositions $x : \tau \vdash \varphi$, $y : \sigma \vdash \psi$, and $z : \rho \vdash \chi$. Let $t : \varphi \longrightarrow \psi$ and $r : \psi \longrightarrow \chi$ be morphisms in $\mathcal{C}\ell$. This means that one can infer $x : \tau \mid \varphi \vdash \psi[t/y]$ and $y : \sigma \mid \psi \vdash \chi[r/z]$. By the substitution and cut rule one gets $x : \tau \mid \varphi \vdash \chi\big[r[t/y] \,/\, z\big]$, so $r[t/y] : \varphi \longrightarrow \chi$ is a morphism in $\mathcal{L}$.

The forgetful functor $\begin{array}{c}\mathcal{L}\\ \downarrow U\\ \mathcal{C}\ell\end{array}$ that sends propositions $\Gamma \vdash \varphi$ to their variable context Γ and morphisms in $\mathcal{L}$ to the underlying morphisms in $\mathcal{C}\ell$, is a fibration. For a context Γ from $\mathcal{C}\ell$, the fibre $\mathcal{L}_\Gamma$ is a preorder category whose objects are the well-formed propositions in context Γ. For a term $x : \tau \vdash t : \sigma$ (which is a morphism $(x : \tau) \longrightarrow (y : \sigma)$ in $\mathcal{C}\ell$) and a proposition $y : \sigma \vdash \psi(y)$ the cartesian morphism over t is t itself regarded as a morphism in $\mathcal{L}$: $\big(x : \tau \vdash \psi[t/y]\big) \longrightarrow \big(y : \sigma \vdash \psi(y)\big)$. The substitution functor for t *does* substitution: it sends $y : \sigma \vdash \psi(y)$ to $x : \tau \vdash \psi[t/y]$. The fibres of $\mathcal{L}$ are bicartesian closed categories if the logic supports the propositional connectives conjunction $\wedge$, disjunction $\vee$, and implication $\supset$ with the usual proof rules (which are omitted for brevity here).

Consider in the following the context $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ and the morphism $(x_1, \dots, x_n) : \Gamma, x : \tau \longrightarrow \Gamma$ in $\mathcal{C}\ell$ that forgets the variable x. Because this morphism can be seen as a projection morphism of a product I denote it with π in the following. The substitution functor π^* maps propositions $\Gamma \vdash \varphi : \mathsf{Prop}$ to $\Gamma, x : \tau \vdash \varphi : \mathsf{Prop}$. This operation of adding a fresh variable to the context is called *(variable context) weakening*. The corresponding derivation rule is as follows.

weakening

$$\frac{\Gamma \mid \varphi \vdash \psi}{\Gamma, x : \tau \mid \varphi \vdash \psi}\ x \notin \Gamma$$

Consider now the proof rules for the quantifiers. It is well known that the usual introduction and elimination rules for universal and existential quantifiers are equivalent to the following two rules (see for instance Lemma 4.1.8. in (Jacobs, 1999a)):

$$\begin{array}{c} \Gamma, x : \tau \mid \varphi \vdash \psi \\ \hline\hline \Gamma \quad \mid \varphi \vdash \forall x : \tau \,.\, \psi \end{array} \ \forall\text{–mate} \qquad\qquad \begin{array}{c} \Gamma, x : \tau \mid \varphi \vdash \psi \\ \hline\hline \Gamma \quad \mid \exists x : \tau \,.\, \varphi \vdash \psi \end{array} \ \exists\text{–mate}$$

(The double line in the preceding rules signals that the rules can be used in both directions upwards and downwards.) These two rules show that universal and existential quantification are right and left adjoints to weakening, respectively.

This example depicted the connection between logic and fibred category theory. Fibrations are built around the notion of substitution. Propositional connectives arise from the cartesian closed structure of the fibre categories. Existential and universal quantification are (respectively) left and right adjoints to weakening functors π^*. ■

For later reference I define two important conditions on fibrations.

Definition 2.3.5 (Beck–Chevalley) Let $\begin{smallmatrix}\mathbb{E}\\ \downarrow\\ \mathbb{B}\end{smallmatrix}$ be a fibration with left adjoints to substitution functors and assume a pullback square in $\mathbb{B}$:

$$\begin{tikzcd}
K \arrow[r, "q"] \arrow[d, "p"'] \arrow[dr, phantom, "\lrcorner", very near start] & L \arrow[d, "g"] \\
I \arrow[r, "f"] & J
\end{tikzcd}$$

The fibration satisfies the *Beck–Chevalley condition* for this pullback, if for any object X in $\mathbb{E}_I$ the (canonical) vertical morphism $\coprod_q p^* X \longrightarrow g^* \coprod_f X$ is an isomorphism in the fibre over L.

Remark 2.3.6 The canonical arrow arises as adjoint transpose of the unit η from the adjunction $\coprod_f \dashv f^*$ in the following way:

$$\begin{array}{cl}
X \xrightarrow{\quad \eta_X \quad} f^* \coprod_f X & \\ \hline
p^* X \longrightarrow (f \circ p)^* \coprod_f X & \text{apply } p^* \\ \hline\hline
p^* X \longrightarrow (g \circ q)^* \coprod_f X & \text{pullback square} \\ \hline\hline
\coprod_q p^* X \longrightarrow g^* \coprod_f X & \coprod_q \dashv q^*
\end{array}$$

For an elaborated discussion of the Beck–Chevalley condition see Section 1.9 of (Jacobs, 1999a).

Definition 2.3.7 (Frobenius) Let $\mathbb{C}$ and $\mathbb{D}$ be categories with binary products that are related by a pair $L \dashv R$ of adjoint functors $L : \mathbb{C} \longrightarrow \mathbb{D}$ and $R : \mathbb{D} \longrightarrow \mathbb{C}$. Assume that the functor R preserves products. The adjunction $L \dashv R$ satisfies the *Frobenius condition* if the following diagram commutes (canonically) up to isomorphism.

$$\begin{tikzcd}
\mathbb{C} \times \mathbb{D} \arrow[r, "\mathsf{Id} \times R"] \arrow[d, "L \times \mathsf{Id}"'] & \mathbb{C} \times \mathbb{C} \arrow[r, "\times"] & \mathbb{C} \arrow[d, "L"] \\
\mathbb{D} \times \mathbb{D} \arrow[rr, "\times"] & & \mathbb{D}
\end{tikzcd}$$

Remark 2.3.8

1. The preceding diagram commutes in **Cat** up to isomorphism if for any objects $X \in |\mathbb{C}|$ and $Y \in |\mathbb{D}|$ there is an isomorphism $L(X \times RY) \cong L(X) \times Y$ in $\mathbb{D}$.

2. The canonical isomorphism arises as adjoint transpose of $\eta \times \mathrm{id}_{RY}$, where η is the unit of the adjunction $L \dashv R$:

$$\begin{array}{cl}
X \times RY \xrightarrow{\eta \times \mathrm{id}} R(LX) \times RY & \\
\hline\hline
X \times RY \longrightarrow R(L(X) \times Y) & R \text{ preserves } \times \\
\hline\hline
L(X \times RY) \longrightarrow L(X) \times Y & L \dashv R
\end{array}$$

3. In this thesis the Frobenius condition is only used when the categories $\mathbb{C}$ and $\mathbb{D}$ are fibres of a fibration and when R is a substitution functor. Let $u : I \longrightarrow J$ be an arrow in the base of the fibration $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$. Then the coproduct along u fulfils the Frobenius condition if, in the fibre over J for arbitrary $X \in \mathbb{E}_I$ and $Y \in \mathbb{E}_J$, it holds that

$$\textstyle\coprod_u (X \wedge u^* Y) \quad \cong \quad (\coprod_u X) \wedge Y$$

Example 2.3.9 The Beck–Chevalley and the Frobenius condition are complicated abstract conditions that can become quite trivial for familiar fibrations. Consider the following pullback square in the category $\mathcal{C}\ell$ from Example 2.2.2. I assume that t is a term of type ρ with one free variable $y : \sigma$.

$$\begin{tikzcd}
(x : \tau) \times (y : \sigma) \arrow[r, "\pi_2"] \arrow[d, "\mathrm{id} \times t"'] \arrow[dr, phantom, very near start, "\lrcorner"] & (y : \sigma) \arrow[d, "t"] \\
(x : \tau) \times (z : \rho) \arrow[r, "\pi_2"] & (z : \rho)
\end{tikzcd}$$

Assume now a proposition $x : \tau,\, z : \rho \vdash \varphi : \mathsf{Prop}$. The Beck–Chevalley condition for this pullback square in the fibration $\begin{smallmatrix}\mathcal{L}\\ \downarrow\\ \mathcal{C}\!\ell\end{smallmatrix}$ is $t^* \coprod_{\pi_2} \varphi \cong \coprod_{\pi_2} (\mathrm{id} \times t)^* \varphi$. If we take into account, that $\coprod_{\pi_2}$ corresponds to existential quantification over x and that t^* is the substitution of t for z we can reformulate the condition. It says that in variable context $y : \sigma$ both propositions $(\exists x : \tau \,.\, \varphi)[t/z]$ and $\exists x : \tau \,.\, (\varphi[t/z, x/x])$ must be provably equivalent. This is indeed the case, because t has no free occurrences of the variable x.

Consider now the adjunction $\coprod_{\pi_2} \dashv \pi_2^*$ for the projection $(x : \tau) \times (z : \rho) \xrightarrow{\pi_2} (z : \rho)$. Assume another proposition $z : \rho \vdash \psi : \mathsf{Prop}$. In the syntax of the logic, the Frobenius condition for this adjunction states the following (recall that π_2^* is weakening, which has no correspondence in the concrete syntax). In variable context $z : \rho$ both propositions $\exists x : \tau \,.\, (\varphi \wedge \psi)$ and $(\exists x : \sigma \,.\, \varphi) \wedge \psi$ must be provable equivalent (which indeed holds because x is not free in ψ). ■

A *morphism between fibrations* $\begin{smallmatrix}\mathbb{D}\\ \downarrow p\\ \mathbb{A}\end{smallmatrix} \longrightarrow \begin{smallmatrix}\mathbb{E}\\ \downarrow q\\ \mathbb{B}\end{smallmatrix}$ is a pair of functors $(H : \mathbb{D} \longrightarrow \mathbb{E}, K : \mathbb{A} \longrightarrow \mathbb{B})$ such that $q \circ H = K \circ p$ and that H sends cartesian morphisms in $\mathbb{D}$ to cartesian morphisms in $\mathbb{E}$. In this case H is also called a *fibred functor over* K or more shortly H *is fibred over* K.

Under the assumption that the equation $q \circ H = K \circ p$ holds the condition that H preserves cartesian morphisms is equivalent to the following statement: For all morphisms $u : I \longrightarrow J$ in $\mathbb{A}$ and objects Y over J we have a canonical vertical isomorphism $\varphi : H(u^* Y) \longrightarrow (K\, u)^* (H\, Y)$ in $\mathbb{E}$. Being canonical means that φ commutes with the liftings, that is that $\widehat{K\, u} \circ \varphi = H\, \widehat{u}$ (which implies $H\, \widehat{u} \circ \varphi^{-1} = \widehat{K\, u}$). In a preorder fibration every vertical isomorphism $H(u^* Y) \cong (K\, u^*)$ is canonical. In proofs that establish fibredness of a functor I will always use the latter condition and construct an isomorphism.

The investigation of the exponent in Subsection 2.4.3 forces me to generalise the notion of a morphism between fibrations to (pairs of) contravariant functors and to cofibrations. These generalisations are not explicit in (Jacobs, 1999a) but they arise in the following way. A pair of functors between two cofibrations is cofibred if it is fibred when regarded as functors between the corresponding fibrations. Further, a pair of contravariant functors is (co–)fibred, if it is (co–)fibred when considered as ordinary functors with opposite domains. The precise definition is as follows.

Definition 2.3.10 Consider two bifibrations $\begin{smallmatrix}\mathbb{D}\\ \downarrow p\\ \mathbb{A}\end{smallmatrix}$ and $\begin{smallmatrix}\mathbb{E}\\ \downarrow q\\ \mathbb{B}\end{smallmatrix}$ (that is $\begin{smallmatrix}\mathbb{D}\\ \downarrow p\\ \mathbb{A}\end{smallmatrix}$ and $\begin{smallmatrix}\mathbb{E}\\ \downarrow q\\ \mathbb{B}\end{smallmatrix}$ are both fibrations and cofibrations).

1. A contravariant functor $H : \mathbb{D}^{\mathrm{op}} \longrightarrow \mathbb{E}$ is *fibred over* $K : \mathbb{A}^{\mathrm{op}} \longrightarrow \mathbb{B}$ if $q \circ H = K \circ p$ and if H sends cartesian morphisms in $\mathbb{D}^{\mathrm{op}}$ (i.e., cocartesian morphisms in $\mathbb{D}$) to cartesian morphisms in $\mathbb{E}$.

2. A covariant functor $H : \mathbb{D} \longrightarrow \mathbb{E}$ is *cofibred over* $K : \mathbb{A} \longrightarrow \mathbb{B}$ if $q \circ H = K \circ p$ and if H sends cocartesian morphisms in $\mathbb{D}$ to cocartesian morphisms in $\mathbb{E}$.

3. A contravariant functor $H : \mathbb{D}^{\mathrm{op}} \longrightarrow \mathbb{E}$ is *cofibred over* $K : \mathbb{A}^{\mathrm{op}} \longrightarrow \mathbb{B}$ if $q \circ H = K \circ p$ and if H sends cocartesian morphisms in $\mathbb{D}^{\mathrm{op}}$ (i.e., cartesian morphisms in $\mathbb{D}$) to cocartesian morphisms in $\mathbb{E}$.

Note that the assumption of two bifibrations is the least common basis of the three definitions. For instance, to define the notion of covariant cofibred functors (which coincides with the notion of morphisms between cofibrations) it is only necessary to assume that $\begin{array}{c}\mathbb{D}\\ \downarrow^{p}\\ \mathbb{A}\end{array}$ and $\begin{array}{c}\mathbb{E}\\ \downarrow^{q}\\ \mathbb{B}\end{array}$ are cofibrations. In this thesis I use the terms of fibredness and cofibredness only with respect to bifibrations.

Lemma 2.3.11 *Let* $\begin{array}{c}\mathbb{D}\\ \downarrow^{p}\\ \mathbb{A}\end{array}$ *and* $\begin{array}{c}\mathbb{E}\\ \downarrow^{q}\\ \mathbb{B}\end{array}$ *be two bifibrations as in the preceding definition and assume that X is an object in $\mathbb{D}$ and that u is a suitable morphism in $\mathbb{A}$. Assume further that for suitable functors K and H the commutation property $q \circ H = K \circ p$ holds.*

1. *A contravariant functor H is fibred over K if and only if there is a canonical isomorphism $(Ku)^* \, (H\,X) \cong H(\coprod_u X)$.*

2. *A covariant functor H is cofibred over K if and only if there is a canonical isomorphism $\coprod_{Ku} (H\,X) \cong H(\coprod_u X)$.*

3. *A contravariant functor H is cofibred over K if and only if there is a canonical isomorphism $\coprod_{Ku} (H\,X) \cong H(u^*\,X)$.* □

Fibredness and cofibredness properties are important for Chapter 3 on binary methods. Especially I have to investigate if, for a bifibration $\begin{array}{c}\mathbb{E}\\ \downarrow\\ \mathbb{B}\end{array}$, the bicartesian closed structure of $\mathbb{E}$ is (co–)fibred over the bicartesian closed structure of $\mathbb{B}$. Let me try to explain what this means in detail.

Recall from Section 2.2 that the cartesian closed structure for a category $\mathbb{C}$ can be given by functors $\mathbb{C} \times \mathbb{C} \longrightarrow \mathbb{C}$ (for product and coproduct) and $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ (for the exponent). Assume in the following a bifibration $\begin{array}{c}\mathbb{E}\\ \downarrow^{p}\\ \mathbb{B}\end{array}$ where both the base and the total category are bicartesian closed. Assume that this bicartesian structure is given by functors $\circledast_{\mathbb{B}} : \mathbb{B} \times \mathbb{B} \longrightarrow \mathbb{B}$ (for the product or the coproduct of the base), $\circledast_{\mathbb{E}} : \mathbb{E} \times \mathbb{E} \longrightarrow \mathbb{E}$ (for the product or the coproduct of the total category), $\ominus_{\mathbb{B}} : \mathbb{B}^{\mathrm{op}} \times \mathbb{B} \longrightarrow \mathbb{B}$ (for the exponent of the base), and finally $\ominus_{\mathbb{E}} : \mathbb{E}^{\mathrm{op}} \times \mathbb{E} \longrightarrow \mathbb{E}$ (for the exponent of the total category). The bicartesian closed structure of $\mathbb{E}$ is (co–)fibred if both $\circledast_{\mathbb{E}}$ is (co–)fibred over $\circledast_{\mathbb{B}}$ and $\ominus_{\mathbb{E}}$ is (co–)fibred over $\ominus_{\mathbb{B}}$.

The product and the coproduct are both given by strictly covariant binary functors. So to find out what it means that $\circledast_{\mathbb{E}}$ is (co–) fibred over $\circledast_{\mathbb{B}}$ we have to double the conditions for covariant fibred and for covariant cofibred functors. We get that $\circledast$ is fibred if

$$p \circ \circledast_{\mathbb{E}} \quad = \quad \circledast_{\mathbb{B}} \circ (p \times p) \tag{2.2}$$

and if for arbitrary morphisms $u : I \longrightarrow J$ and $v : K \longrightarrow L$ in the base $\mathbb{B}$ and for all objects X over J and Y over L there is an isomorphism

$$(u \circledast_{\mathbb{B}} v)^{*} \, (X \circledast_{\mathbb{E}} Y) \quad \cong \quad (u^{*} \, X) \circledast_{\mathbb{E}} (v^{*} \, Y) \tag{2.3}$$

The functor $\circledast_{\mathbb{E}}$ is cofibred over $\circledast_{\mathbb{B}}$ if $p, \circledast_{\mathbb{E}}$, and $\circledast_{\mathbb{B}}$ commute as before. This time the required isomorphism is

$$\textstyle\coprod_{u \circledast_{\mathbb{B}} v} (U \circledast_{\mathbb{E}} V) \quad \cong \quad (\coprod_{u} U) \circledast_{\mathbb{E}} (\coprod_{v} V)$$

Here, $u : I \longrightarrow J$ and $v : K \longrightarrow L$ are arbitrary morphisms in $\mathbb{B}$ as before and U and V are objects over (respectively) I and K.

For the exponent the situation is more difficult, because it is contravariant in its first argument and covariant in its second argument. So to find out what it means for $\ominus_{\mathbb{E}}$ to be (co–) fibred over $\ominus_{\mathbb{B}}$ we have to apply the definition for contravariant (co–) fibred functors to the first position and that of covariant (co–) fibred functors to the second position. Again the basic requirement is that $\ominus$ and p commute:

$$p \circ \ominus_{\mathbb{E}} \quad = \quad \ominus_{\mathbb{B}} \circ (p \times p)$$

Because of the contravariant first argument we have to be careful to turn the right morphisms around. Assume two arrows $u : I \longrightarrow J$ and $v : K \longrightarrow L$ in the base category. Then $u \ominus_{\mathbb{B}} v$ is an arrow $J \ominus_{\mathbb{B}} K \longrightarrow I \ominus_{\mathbb{B}} L$. The exponent $\ominus_{\mathbb{E}}$ is fibred over $\ominus_{\mathbb{B}}$ if for all such morphisms u and v in the base and for all objects U over I and Y over L there is an isomorphism

$$(u \ominus_{\mathbb{B}} v)^{*} \, (U \ominus_{\mathbb{E}} Y) \quad \cong \quad (\textstyle\coprod_{u} U) \ominus_{\mathbb{E}} (v^{*} \, Y)$$

in the fibre over $J \ominus_{\mathbb{B}} K$.

As a last variation of this theme, $\ominus_{\mathbb{E}}$ is cofibred over $\ominus_{\mathbb{B}}$ if for all morphisms u and v as before and for all objects X over J and V over K there is an isomorphism

$$\textstyle\coprod_{(u \ominus_{\mathbb{B}} v)} (X \ominus_{\mathbb{E}} V) \quad \cong \quad (u^{*} \, X) \ominus_{\mathbb{E}} (\coprod_{v} V)$$

in the fibre over $I \ominus_{\mathbb{B}} L$.

Example 2.3.12 The Sections 2.4.1 and 2.4.2 (starting on page 35) investigate all combinations of co– and contravariant (co–) fibredness for two concrete bifibrations: typed predicates and typed relations.

Here I continue the syntactic example of the fibration $\begin{smallmatrix}\mathcal{L}\\ \downarrow\\ \mathcal{C}\ell\end{smallmatrix}$. Consider the functor $\times_{\mathcal{L}} : \mathcal{L} \times \mathcal{L} \longrightarrow \mathcal{L}$. It sends two predicates $\Gamma \vdash \varphi$ and $\Delta \vdash \psi$ to their conjunction $\Gamma, \overline{\Delta} \vdash \varphi \wedge \overline{\psi}$ where $\overline{\Delta}$ and $\overline{\psi}$ arise from (respectively) Δ and ψ via a renaming of the variables in Δ. The renaming is necessary to make the concatenation of Γ and Δ a context with distinct variables again. The functor $\times_{\mathcal{L}}$ defines products in the total category $\mathcal{L}$.

Now let us look, what it means for $\times_{\mathcal{L}}$ to be a fibred functor (over the product in $\mathcal{C}\ell$). Because the product of two contexts in $\mathcal{C}\ell$ is their concatenation, Equation 2.2 states that also $\times_{\mathcal{L}}$ must concatenate the contexts of the predicates. Additionally Equation 2.3 must hold. To reformulate this equation in the syntax of the logic, assume four contexts Γ, Γ', Δ, and Δ' where $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ and $\Delta = y_1 : \sigma_1, \dots, y_m : \sigma_m$. Assume further two predicates $\Gamma \vdash \varphi$ and $\Delta \vdash \psi$ and two morphism $(t_1, \dots, t_n) : \Gamma' \longrightarrow \Gamma$ and $(s_1, \dots, s_m) : \Delta' \longrightarrow \Delta$ in $\mathcal{C}\ell$. Equation 2.3 states that for all such predicates and terms the two predicates

$$\Gamma', \Delta' \quad \vdash \quad (\varphi \wedge \psi)[t_1/x_1, \dots, t_n/x_n,\ s_1/y_1, \dots, s_m/y_m]$$

and

$$\Gamma', \Delta' \quad \vdash \quad (\varphi[t_1/x_1, \dots, t_n/x_n]) \ \wedge\ (\psi[s_1/y_1, \dots, s_m/y_m])$$

are provably equivalent (which is indeed the case). ∎

2.4. Predicate Logic over Sets

This section introduces two fibrations in detail. Namely the fibration of typed predicates and that of typed relations. Let me explain why I decided to work with the two concrete fibrations $\begin{smallmatrix}\mathbf{Pred}\\ \downarrow\\ \mathbf{Set}\end{smallmatrix}$ and $\begin{smallmatrix}\mathbf{Rel}\\ \downarrow\\ \mathbf{Set}\times\mathbf{Set}\end{smallmatrix}$ instead of assuming an arbitrary bifibration $\begin{smallmatrix}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{smallmatrix}$ with additional properties.

Indeed the whole development of this section (and also of Chapter 3 where I extend the notion of coalgebras to allow for binary methods) could be done for an arbitrary bifibration $\begin{smallmatrix}\mathbb{E}\\ \downarrow\\ \mathbb{B}\end{smallmatrix}$. In this case the fibration of relations over $\mathbb{E}$ is constructed by *change of base*. This means we form the following pullback in **Cat**.

$$\begin{tikzcd}
\mathbf{Rel}(\mathbb{E}) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbb{E} \arrow[d, "p"] \\
\mathbb{B} \times \mathbb{B} \arrow[r, "\times"'] & \mathbb{B}
\end{tikzcd} \tag{2.4}$$

By general considerations (compare Section 1.5 of (Jacobs, 1999a)) it follows that the functor $\begin{smallmatrix}\mathbf{Rel}(\mathbb{E})\\ \downarrow\\ \mathbb{B}\times\mathbb{B}\end{smallmatrix}$ obtained this way is a fibration again. For the results of this section one has to assume extra properties. For instance that $\mathbb{B}$ is bicartesian closed and distributive, that the coproducts in $\mathbb{B}$ are distributive, that the fibres of $\mathbb{E}$ have bicartesian closed structure (which is preserved by the substitution functors), that the Beck–Chevalley and the Frobenius conditions hold, and so on. In fact most of the results of this section have been obtained this way by (Hermida and Jacobs, 1998) and (Hensel, 1999) for an arbitrary fibration $\begin{smallmatrix}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{smallmatrix}$ with additional properties. The proofs are quite technical and require a nontrivial amount of fibred category theory.

For the treatment of binary methods I have to consider cofibredness in addition to the properties that are discussed in (Hermida and Jacobs, 1998) and in (Hensel, 1999). The proof of Lemma 2.4.17 (establishing a restricted cofibredness property for the bicartesian closed structure of the total category) for an arbitrary fibration requires additionally that the Axiom of Choice holds in this fibration. So even when working in an abstract setting, the results depend on quite strong assumptions. The greater difficulty when working with an arbitrary fibration hardly justifies the small additional level of generality that can be obtained.

In coalgebraic specification one is mainly interested in models in **Set**. In this thesis I give a semantics in **Set** for the specification language CCSL (Chapter 4). Applications of CCSL use currently a semantics in the higher-order logics of the theorem provers PVS (see (Owre et al., 1996; Owre et al., 1995)) or ISABELLE/HOL (see (Nipkow et al., 2002b)). Both environments differ only very little from **Set**.

For these reasons it is more appropriate to use the two concrete fibrations of predicates and relations over **Set** for the Chapter on binary methods in this thesis. Fibred category theory will provide the appropriate language to express the relevant properties of predicates and relations. Therefore this section introduces predicates and relations as fibrations over **Set** and proves all the properties that are needed in Chapter 3. The fibrations of predicates and relations are primary examples for fibred category theory, so they are discussed in detail for instance in (Jacobs, 1999a) and (Hensel, 1999). Almost all results in this section are either folklore or appear somewhere in the literature, sometimes in more general form. Only the question of cofibredness (Example 2.4.16 and Propositions 2.4.17, page 46f) has, to the best of my knowledge, not been investigated before.

All results of this section have been formalised and proved with PVS. The last subsection (starting on page 48) explains some aspects of the PVS formalisation. A detailed explanation of the PVS material and a correspondence between the lemmas of this section and the PVS formalisation is in Appendix A.

2.4.1. The Fibration of Predicates

The fibration of typed predicates arises as the subobject fibration of the category **Set**, see Section 1.3 in (Jacobs, 1999a). Here I prefer to give a concrete definition.

Definition 2.4.1 (Fibration of Predicates) The category of typed predicates is denoted by **Pred**. Its objects are pairs of sets, written as $(P \subseteq X)$, such that P is a subset of X. Morphisms in **Pred** are ordinary functions: A function $f : X \longrightarrow Y$ is a morphism $f : (P \subseteq X) \longrightarrow (Q \subseteq Y)$ in **Pred** if $f\,x \in Q$ whenever $x \in P$. The identities in **Pred** are the identity functions and composition in **Pred** is functional composition. The category **Pred** is fibred over **Set** via the forgetful functor that sends a predicate $(P \subseteq X)$ to its carrier X and morphisms in **Pred** to the underlying functions.

For a predicate $(P \subseteq X)$ I sometimes write $P\,x$ instead of $x \in P$ to denote that the predicate holds for the individual $x \in X$. For the objects of **Pred** I use letters P, Q to emphasise that they are predicates. The forgetful functor $\begin{smallmatrix}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{smallmatrix}$ is a bifibration. For a set X, the fibre over X contains all subsets of X. The vertical morphisms are inclusions. Assume a function $f : X \longrightarrow Y$ in the following. For a predicate $(Q \subseteq Y)$ the cartesian morphism over f is $f : (\{x \mid f\,x \in Q\} \subseteq X) \longrightarrow (Q \subseteq Y)$. For a predicate $(P \subseteq X)$ the cocartesian morphism over f is $f : (P \subseteq X) \longrightarrow (\{f\,x \mid x \in P\} \subseteq Y)$. The substitution functor f^* is given by $f^*\,(Q \subseteq Y) = (\{x \mid f\,x \in Q\} \subseteq X)$. Note that you can consider a morphism $f : (P \subseteq X) \longrightarrow (Q \subseteq Y)$ as a pair that consists of a function $f : X \longrightarrow Y$ and a proof of the sequent $x : X \mid P\,x \vdash Q(f\,x)$.

Before I discuss the structure of the predicate fibration let me say something about notation. Soon I will deal with three different bicartesian structures: The one in the fibres of **Pred**, the one in the total category **Pred**, and the one in the base category **Set**. The same will happen for **Rel**, the fibration of relations (see below, Definition 2.4.3 on page 38). So it is a good idea to use notation that distinguishes all the different products, coproducts and exponents. For the bicartesian structure in **Set** I use the usual symbols $\times, +$, and $\Rightarrow$. For the total categories I annotate these symbols with the subscript $-_{\mathrm{P}}$ for **Pred** and $-_{\mathrm{R}}$ for **Rel**. So for instance $\times_{\mathrm{P}}$ is the cartesian product in **Pred**. Because its very close relationship with the corresponding logical operation, I use $\wedge, \vee$, and $\supset$ for the product, the coproduct, and the exponent (implication) in the fibres of **Pred** and **Rel**.

The fibration $\begin{smallmatrix}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{smallmatrix}$ has a very rich structure. Each fibre is bicartesian closed:

$$\begin{array}{rcccl}
(P \subseteq X) \wedge (Q \subseteq X) & = & (P \cap Q \subseteq X) & = & (\{x \mid P\,x \text{ and } Q\,x\} \subseteq X) \\
(P \subseteq X) \vee (Q \subseteq X) & = & (P \cup Q \subseteq X) & = & (\{x \mid P\,x \text{ or } Q\,x\} \subseteq X) \\
(P \subseteq X) \supset (Q \subseteq X) & = & ((X \setminus P) \cup Q \subseteq X) & = & (\{x \mid P\,x \text{ implies } Q\,x\} \subseteq X)
\end{array}$$

Moreover, this bicartesian structure is fibred, that is, it is preserved by the substitution functors. The bicartesian structure on morphisms is trivial because the fibres of **Pred** are preorder categories, that is, there is at most one vertical morphism between any two objects in **Pred**. Again I invite those readers not familiar with category theory, to check that the required properties hold.

The product and the coproduct in the fibres can be generalised to arbitrary collections. Let I be an arbitrary index set and let $(P_i \subseteq X)_{i \in I}$ be an I–indexed collection of predicates:

$$\begin{array}{rclcl}
\bigwedge_{i \in I} (P_i \subseteq X) & = & \bigcap_{i \in I} P_i & = & (\{x \mid \forall i \in I \,.\, P_i\, x\} \subseteq X) \\
\bigvee_{i \in I} (P_i \subseteq X) & = & \bigcup_{i \in I} P_i & = & (\{x \mid \exists i \in I \,.\, P_i\, x\} \subseteq X)
\end{array}$$

Also these generalised products and coproducts are fibred, that is we have $u^* \left(\bigwedge_i P_i\right) = \bigwedge_i (u^*\, P_i)$ and $u^* \left(\bigvee_i P_i\right) = \bigvee_i (u^*\, P_i)$.

In the fibration $\begin{array}{c}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{array}$ left and right adjoints exist to all substitution functors. I denote them by $\coprod_f$ and $\prod_f$ for a function $f : X \longrightarrow Y$. They are explicitly given by[3]

$$\begin{array}{rcl}
\coprod_f (P \subseteq X) & = & \big(\{y \mid \exists x \in X \,.\, f\, x = y \;\text{ and }\; x \in P\} \subseteq Y\big) \\
\prod_f (P \subseteq X) & = & \big(\{y \mid \forall x \in X \,.\, f\, x = y \;\text{ implies }\; x \in P\} \subseteq Y\big)
\end{array}$$

Remark 2.4.2 Let me stress the fact that we have now enough structure in $\begin{array}{c}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{array}$ to use it as semantic universe for a simply typed predicate logic (compare Example 2.3.4). Sequents of simply typed predicate logic have the form $\Gamma \vdash \varphi$ where Γ is the variable context of φ. Formulae are build up from atomic predicates $(x : \tau \vdash P\, x)$, logical connectives ($\wedge, \vee$, and $\supset$) and quantification.

A semantics for this logic will assign a set $\llbracket \Gamma \rrbracket$ to every context Γ and a predicate in the fibre over $\llbracket \Gamma \rrbracket$ to every formula $\Gamma \vdash \varphi$. Assume we are given a set $\llbracket \tau \rrbracket$ for each type τ. The semantics for contexts is $\llbracket x_1 : \tau_1, \dots, x_n : \tau_n \rrbracket = \llbracket \tau_1 \rrbracket \times \cdots \times \llbracket \tau_n \rrbracket$. Assume further, we have also a predicate $(\llbracket P \rrbracket \subseteq \llbracket \tau \rrbracket)$ for each atomic proposition $x : \tau \vdash P\, x$ of the logic. Then the bicartesian structure in the fibres gives a semantics to the boolean connectives. The substitution functors of $\begin{array}{c}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{array}$ provide the desired semantics for syntactic substitution. The more interesting question is how to deal with weakening and quantification.

Weakening adds a fresh variable to the context: If $\Gamma \vdash \varphi$ is formula, then also $\Gamma, x : \tau \vdash \varphi$ is a formula, provided x is not already contained in Γ. On the semantic side we have a projection $\llbracket \Gamma \rrbracket \times \llbracket \tau \rrbracket \xrightarrow{\pi} \llbracket \Gamma \rrbracket$. The substitution functor π^* sends predicates over $\llbracket \Gamma \rrbracket$ to predicates over $\llbracket \Gamma \rrbracket \times \llbracket \tau \rrbracket$. It behaves exactly in the expected way to give a semantics to weakening.

Quantification is somehow inverse to weakening: If $\Gamma, x : \tau \vdash \varphi$ is a formula, then so is $\Gamma \vdash \exists x : \tau \,.\, \varphi$. In Example 2.3.4 I showed that existential and universal quantification is (respectively) left and right adjoint to weakening. Therefore we get as semantics of the quantifiers $\llbracket \exists x : \tau \,.\, \varphi \rrbracket = \coprod_\pi \llbracket \varphi \rrbracket$ and $\llbracket \forall x : \tau \,.\, \varphi \rrbracket = \prod_\pi \llbracket \varphi \rrbracket$.

[3] The operation $\coprod_f$ is well known as the image of f, sometimes denoted as $f[-]$. However, I prefer the notation from fibred category theory. The same applies to f^*, sometimes written as $f^{-1}(-)$.

There is also a bicartesian closed structure in the total category **Pred**. Assume two predicates $(P \subseteq X)$ and $(Q \subseteq Y)$, then

$$\begin{array}{rcl}
(P \times_{\mathrm{P}} Q \ \subseteq\ X \times Y) & = & (\pi_1^* \, P) \ \wedge \ (\pi_2^* \, Q) \\
 & = & \big(\{(x,y) \mid P\,x \ \text{ and } \ Q\,y\} \ \subseteq\ X \times Y\big) \\
(P +_{\mathrm{P}} Q \ \subseteq\ X + Y) & = & (\coprod_{\kappa_1} P) \ \vee \ (\coprod_{\kappa_2} Q) \\
 & = & \big(\{\kappa_1\, x \mid P\,x\} \ \cup \ \{\kappa_2\, y \mid Q\,y\} \ \subseteq\ X + Y\big) \\
(P \Rightarrow_{\mathrm{P}} Q \ \subseteq\ X \Rightarrow Y) & = & \prod_{\pi_1} \big((\pi_2^* \, P) \ \supset \ (\mathsf{eval}_{X,Y}{}^* \, Q)\big) \\
 & = & \big(\{f \mid \forall x \in X \,.\, P\,x \ \text{ implies } \ Q(f\,x)\} \ \subseteq\ X \Rightarrow Y\big)
\end{array}$$

The fibred exponent $\supset$ in the equation above is the one in the fibre over $(X \Rightarrow Y) \times X$. The projection functions are $X \Rightarrow Y \xleftarrow{\pi_1} (X \Rightarrow Y) \times X \xrightarrow{\pi_2} X$. The equations defining the bicartesian closed structure in terms of the fibred structure are not a coincidence: They follow from general properties of the category **Set** and the fibres of **Pred**, compare (Hermida and Jacobs, 1998). On morphisms the bicartesian structure is inherited from the base category, so $f \odot_{\mathrm{P}} g \ = \ f \odot g$ for $\odot \in \{\times, +, \Rightarrow\}$ and suitable morphisms f and g in **Pred**.

All fibres of **Pred** have both initial and final objects. For a set X the final object in the fibre over X is the constantly true predicate $\top_X = (X \subseteq X)$. The initial object is the empty predicate $\bot_X = (\emptyset \subseteq X)$. The final objects of the fibres define the truth functor $\mathbf{1} : \mathbf{Set} \longrightarrow \mathbf{Pred}$ that assigns to every set X the truth predicate over it. For the predicate fibration $\begin{array}{c}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{array}$ the functor U is left adjoint to $\mathbf{1}$.

The truth functor also has a right adjoint. Thereby the predicate fibration admits *comprehension* in the sense of (Lawvere, 1970). This right adjoint comprehension $\mathbf{1} \dashv \{-\}$ is explicitly given by $\{(P \subseteq X)\} = P$.

2.4.2. The Fibration of Relations

Let me introduce the fibration of relations before I continue to discuss the properties of the predicate fibration. Any relation can be regarded as a binary predicate. Therefore the fibration of relations can be obtained by change of base, see Diagram 2.4 at the beginning of this section. For illustration I give an explicit description.

Definition 2.4.3 (Fibration of Relations) The category of relations **Rel** has as objects triples, written as $(R \subseteq X \times Y)$, such that R is a relation over X and Y. Instead of $x\,R\,y$ I often write $R(x, y)$ to denote that R relates the two individuals x and y. Morphisms in **Rel** are pairs of functions. The functions $f : U \longrightarrow X$ and $g : V \longrightarrow Y$ form a morphism $(f, g) : (S \subseteq U \times V) \longrightarrow (R \subseteq X \times Y)$ if for all u and v with $u\,S\,v$ it holds that $(f\,u)\,R\,(g\,v)$. Identities and composition are given by the identities and (pairwise) composition of functions in **Set**.

The category **Rel** is fibred over **Set** × **Set** via the forgetful functor that sends a relation $(R \subseteq X \times Y)$ to the object (X, Y) in **Set** × **Set**.

Remark 2.4.4 In (Jacobs, 1999a) and in (Hermida and Jacobs, 1998) the fibration of relations is defined via the pullback

$$\begin{tikzcd}
\mathbf{SRel}(\mathbb{E}) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathbb{E} \arrow[d, "p"] \\
\mathbb{B} \arrow[r, "I \mapsto I \times I"'] & \mathbb{B}
\end{tikzcd}$$

If we instantiate $\overset{\mathbb{E}}{\underset{\mathbb{B}}{\downarrow p}}$ with the predicate fibration we get the category **SRel**(**Pred**) of *single carrier binary relations* $(R \subseteq X \times X)$ (instead of $(R \subseteq X \times Y)$ as in preceding definition). Obviously, the category **SRel**($\mathbb{E}$) is a subcategory of **Rel**($\mathbb{E}$). If $\mathbb{B}$ is a distributive category and if coproduct functors exist along the coproduct injections then the inclusion functor $\mathbf{SRel} \hookrightarrow \mathbf{Rel}$ has both a left and a right adjoint (Hensel, 1999). When defining bisimulations for coalgebras it is more natural to work in the category **Rel** with relations on different carriers. On the other hand the notion of quotients makes only sense for binary relations on one carrier.

It is obvious that the fibre of **Rel** over (X, Y) (assuming arbitrary sets X and Y) coincides with the fibre of **Pred** over $X \times Y$. Similarly any morphism (f, g) in **Rel** is a morphism $f \times g$ in **Pred**. So also **Rel** has fibred bicartesian structure. It is identical with the structure in the corresponding fibres in **Pred** (this justifies the use of the same notation). For convenience I repeat the definitions. Assume two relations $R \subseteq X \times Y$ and $S \subseteq X \times Y$, then

$$\begin{array}{rcl}
(R \subseteq X \times Y) \wedge (S \subseteq X \times Y) & = & (R \cap S \subseteq X \times Y) \\
 & = & (\{(x,y) \mid R(x,y) \text{ and } S(x,y)\} \subseteq X \times Y) \\
(R \subseteq X \times Y) \vee (S \subseteq X \times Y) & = & (R \cup S \subseteq X \times Y) \\
 & = & (\{(x,y) \mid R(x,y) \text{ or } S(x,y)\} \subseteq X \times Y) \\
(R \subseteq X \times Y) \supset (S \subseteq X \times Y) & = & ((X \times Y \setminus R) \cup S \subseteq X \times Y) \\
 & = & (\{(x,y) \mid R(x,y) \text{ implies } S(x,y)\} \subseteq X \times Y)
\end{array}$$

The product and the coproduct generalise to arbitrary collections. Let $(R_i \subseteq X \times Y)$ be an arbitrary I–indexed collection of relations:

$$\begin{array}{rcl}
\bigwedge_{i \in I} (R_i \subseteq X \times Y) & = & \bigcap_{i \in I} R_i \\
 & = & (\{(x,y) \mid \forall i \in I \,.\, R_i(x,y)\} \subseteq X \times Y) \\
\bigvee_{i \in I} (R_i \subseteq X \times Y) & = & \bigcup_{i \in I} R_i \\
 & = & (\{(x,y) \mid \exists i \in I \,.\, R_i(x,y)\} \subseteq X \times Y)
\end{array}$$

The fibred bicartesian structure for morphisms is again trivial. The substitution, coproduct and product functors are derived from the ones in **Pred**. Assume two functions $f : U \longrightarrow X$ and $g : V \longrightarrow Y$, and two relations $S \subseteq U \times V$, and $R \subseteq X \times Y$:

$$
\begin{aligned}
(f,g)^*\,(R \subseteq X \times Y) &= (f \times g)^*\,(R \subseteq X \times Y) \\
&= \big(\{(u,v) \mid R(f\,u, g\,v)\} \;\subseteq\; U \times V\big) \\
\textstyle\coprod_{(f,g)}\,(S \subseteq U \times V) &= \textstyle\coprod_{f\times g}\,(S \subseteq U \times V) \\
&= \big(\{(x,y) \mid \exists u \in U, v \in V \,.\, f\,u = x \,\wedge \\
&\qquad\qquad g\,v = y \;\wedge\; S(u,v)\} \;\subseteq\; X \times Y\big) \\
\textstyle\prod_{(f,g)}\,(S \subseteq U \times V) &= \textstyle\prod_{f\times g}\,(S \subseteq U \times V) \\
&= \big(\{(x,y) \mid \forall u \in U, v \in V \,.\, f\,u = x \,\wedge \\
&\qquad\qquad g\,v = y \;\text{ implies }\; S(x,y)\} \;\subseteq\; X \times Y\big)
\end{aligned}
$$

In the following, when working in **Rel**, I often silently switch to the corresponding fibres in **Pred** and also use the notation from **Pred**. So I usually write $(f \times g)^*$ instead of $(f,g)^*$. With the substitution functors it is clear, which morphisms in **Rel** are cartesian: For the pair (f,g) and R from before, the cartesian morphism is $(f,g) : (f \times g)^*\,R \longrightarrow R$.

Next I present the bicartesian closed structure of **Rel**. Assume now two relations $S \subseteq U \times V$ and $R \subseteq X \times Y$. The structure is given by

$$
\begin{aligned}
S \times_{\mathrm{R}} R \;\subseteq\; & (U \times X) \times (V \times Y) \\
&= \; ((\pi_1 \times \pi_1)^*\,S) \;\wedge\; ((\pi_2 \times \pi_2)^*\,R) \\
&= \; \big\{\big((u,x),(v,y)\big) \mid S(u,v) \text{ and } R(x,y)\big\} \\
S +_{\mathrm{R}} R \;\subseteq\; & (U + X) \times (V + Y) \\
&= \; (\textstyle\coprod_{\kappa_1\times\kappa_1} S) \;\vee\; (\textstyle\coprod_{\kappa_2\times\kappa_2} R) \\
&= \; \big\{(\kappa_1\,u, \kappa_1\,v) \mid S(u,v)\big\} \;\cup\; \big\{(\kappa_2\,x, \kappa_2\,y) \mid R(x,y)\big\} \\
S \Rightarrow_{\mathrm{R}} R \;\subseteq\; & (U \Rightarrow X) \times (V \Rightarrow Y) \\
&= \; \textstyle\prod_{\pi_1\times\pi_1} \big((\pi_2 \times \pi_2)^*\,S \;\supset\; (\mathsf{eval}_{U,X} \times \mathsf{eval}_{V,Y})^*\,R\big) \\
&= \; \big\{(f,g) \mid \forall u \in U, v \in V \,.\, S(u,v) \;\text{ implies }\; R(f\,u, g\,v)\big\}
\end{aligned}
$$

The fibred implication $\supset$ above is in the fibre over $(U \Rightarrow X) \times U \;\times\; (V \Rightarrow Y) \times V$. The product and substitution functors sit over the following diagram in the base.

$$\begin{tikzcd}
 & & X \times Y \\
(U \Rightarrow X) \times (V \Rightarrow Y) & (U \Rightarrow X) \times U \;\times\; (V \Rightarrow Y) \times V \arrow[l, "\pi_1\times\pi_1"'] \arrow[r, "\pi_2\times\pi_2"] \arrow[ur, "\mathsf{eval}_{U,X}\times\mathsf{eval}_{V,Y}"] & U \times V
\end{tikzcd}$$

Again the structure on morphisms is (almost) inherited from the base category **Set**: $(u, v) \odot_{\mathrm{R}} (f, g) \;=\; ((u \odot f) \times (v \odot g))$ for $\odot \in \{\times, +, \Rightarrow\}$ and suitable functions u, v, f and g.

The fibration of relations inherits the truth functor from $\begin{smallmatrix}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{smallmatrix}$ and thereby also its adjoint comprehension $\{-\}$. Moreover, the fibration of relations admits equality in the sense of (Lawvere, 1970): Because the base category **Set** has binary products we can form the composite functor $\mathrm{Eq} \overset{\mathrm{def}}{=} \coprod_\delta \circ \mathbf{1} : \mathbf{Set} \longrightarrow \mathbf{Rel}$. Here $\coprod_\delta$ is the coproduct along the diagonal $\delta \overset{\mathrm{def}}{=} \langle \mathrm{id}_X, \mathrm{id}_X \rangle : X \longrightarrow X \times X$. It extends to a functor $\coprod_\delta : \mathbf{Pred} \longrightarrow \mathbf{Rel}$. The composite Eq gives us (typed) equality: $\mathrm{Eq}(X) = (\{(x, x) \mid x \in X\} \subseteq X \times X)$. By composition equality has a right adjoint $\mathrm{Eq} \dashv \{-\} \circ \delta^*$.

Note that equality can be more precisely defined as functor $\mathbf{Set} \longrightarrow \mathbf{SRel}$. In this case *quotients* arise as left adjoint to equality (Jacobs, 1999a). Because the inclusion $\mathbf{SRel} \hookrightarrow \mathbf{Rel}$ has a left adjoint, also equality when considered as a functor $\mathbf{Set} \longrightarrow \mathbf{Rel}$ has a left adjoint. This left adjoint takes a relation $R \subseteq X \times Y$, turns it into an equivalence relation $\widehat{R}$ on $X + Y$ and yields the quotient $(X + Y)/\widehat{R}$.

Swapping all pairs in a relation $R \subseteq X \times Y$ yields the opposite relation R^{op}. More formally:

$$R^{\mathrm{op}} \subseteq Y \times X \quad = \quad \langle \pi_2, \pi_1 \rangle^* R \quad = \quad \big(\{(y, x) \mid x\, R\, y\} \subseteq Y \times X\big)$$

As last example of what can be captured fibrationally I define the composition of two relations in **Rel**. Assume two relations $S \subseteq U \times X$ and $R \subseteq X \times Y$.

$$\begin{aligned} S \circ R \subseteq U \times Y \quad &= \quad \textstyle\coprod_{\pi_2} (\pi_1^* S \;\wedge\; \pi_3^* R) \\ &= \quad \big(\{(u, y) \mid \exists x : X \,.\, S(u, x) \;\text{ and }\; R(x, y)\} \subseteq U \times Y\big) \end{aligned}$$

This construction takes place over the following diagram in the base.

$$\begin{tikzcd} U \times X & U \times X \times Y \arrow[l, "\pi_1"'] \arrow[r, "\pi_3"] \arrow[d, "\pi_2"'] & X \times Y \\ & U \times Y & \end{tikzcd}$$

2.4.3. Properties of Predicates and Relations

The remainder of this section presents technical results about the fibrations of predicates and that of relations. Some of them are *very* trivial. I state them nevertheless, because I refer to these results in Section 2.6 and in Chapter 3. Most of the following results have been obtained by (Hermida and Jacobs, 1998) and (Hensel and Jacobs, 1997) on a more abstract level. As I noted before, this whole section has been formalised within PVS. This applies especially to all lemmas of this subsection. They have all been proved within PVS. For details see the next subsection (starting on page 48).

The following is an immediate consequence of the fact that the bicartesian operations both on **Pred** and on **Rel** are functors and thus preserve entailment.

Lemma 2.4.5 (Monotonicity)

1. *Assume four predicates* $P_1, P_2 \subseteq X$ *and* $Q_1, Q_2 \subseteq Y$. *Then* $P_1 \subseteq P_2$ *and* $Q_1 \subseteq Q_2$ *implies*

$$\begin{aligned} P_1 \times_{\mathrm{P}} Q_1 &\subseteq P_2 \times_{\mathrm{P}} Q_2 \\ P_1 +_{\mathrm{P}} Q_1 &\subseteq P_2 +_{\mathrm{P}} Q_2 \\ P_2 \Rightarrow_{\mathrm{P}} Q_1 &\subseteq P_1 \Rightarrow_{\mathrm{P}} Q_2 \end{aligned}$$

2. *Assume four relations* $S_1, S_2 \subseteq U \times V$ *and* $R_1, R_2 \subseteq X \times Y$. *Then* $S_1 \subseteq S_2$ *and* $R_1 \subseteq R_2$ *implies*

$$\begin{aligned} S_1 \times_{\mathrm{R}} R_1 &\subseteq S_2 \times_{\mathrm{R}} R_2 \\ S_1 +_{\mathrm{R}} R_1 &\subseteq S_2 +_{\mathrm{R}} R_2 \\ S_2 \Rightarrow_{\mathrm{R}} R_1 &\subseteq S_1 \Rightarrow_{\mathrm{R}} R_2 \end{aligned}$$

□

Note that both exponents are anti–monotone in their contravariant position.

Lemma 2.4.6 (Beck–Chevalley and Frobenius)

1. *The Beck–Chevalley condition holds in both fibrations* $\begin{array}{c}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{array}$ *and* $\begin{array}{c}\mathbf{Rel}\\ \downarrow U\\ \mathbf{Set}\times\mathbf{Set}\end{array}$ *for all pullback diagrams in* **Set** *and* **Set** × **Set**, *respectively.*

2. *The Frobenius condition holds in both fibrations for left adjoints to substitution functors. That is, for the predicate fibration Frobenius holds for all* $\coprod_f \dashv f^*$ *and for the fibration of relation it holds for all* $\coprod_{f\times g} \dashv (f \times g)^*$.

Proof This lemma has been proved in PVS. The parts for the fibration of relations follow trivially from the corresponding properties of the predicate fibration. For the latter assume a pullback square

$$\begin{tikzcd} X \arrow[r, "u"] \arrow[d, "v"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "g"] \\ B \arrow[r, "f"] & C \end{tikzcd}$$

Then for Item (1) we have to prove the following equation for a predicate ($P \subseteq B$).

$$\coprod_u v^* P \quad = \quad g^* \coprod_f P$$

For an additional predicate ($Q \subseteq C$) Item (2) corresponds to

$$\coprod_f (P \wedge f^* Q) \quad = \quad (\coprod_f P) \wedge Q$$

Both equations are easy computations. □

In the next lemma about the truth functor, the first two equation follow from the fact that truth is both a left and a right adjoint.

Lemma 2.4.7 (Truth) *For arbitrary sets X and Y*

$$\begin{array}{rcl}\top_X \times_{\mathrm{P}} \top_Y & = & \top_{X\times Y} \\ \top_X +_{\mathrm{P}} \top_Y & = & \top_{X+Y} \\ \top_X \Rightarrow_{\mathrm{P}} \top_Y & = & \top_{X\Rightarrow Y}\end{array}$$

Proof Immediate from the definition. □

Lemma 2.4.8 (Equality) *For arbitrary sets X and Y*

$$\begin{array}{rcl}\mathrm{Eq}(X) \times_{\mathrm{R}} \mathrm{Eq}(Y) & = & \mathrm{Eq}(X \times Y) \\ \mathrm{Eq}(X) +_{\mathrm{R}} \mathrm{Eq}(Y) & = & \mathrm{Eq}(X + Y) \\ \mathrm{Eq}(X) \Rightarrow_{\mathrm{R}} \mathrm{Eq}(Y) & = & \mathrm{Eq}(X \Rightarrow Y)\end{array}$$

Proof Straightforward. □

Lemma 2.4.9 (Conjunction) *Let A be a set and I be an arbitrary index set.*

1. *Assume collections $(P_i \subseteq X)_{i\in I}$ and $(Q_i \subseteq Y)_{i\in I}$ of predicates. Then*

$$\begin{array}{rcl}\bigwedge_i (P_i \times_{\mathrm{P}} Q_i) & = & \bigwedge_i P_i \times_{\mathrm{P}} \bigwedge_i Q_i \\ \bigwedge_i (P_i +_{\mathrm{P}} Q_i) & = & \bigwedge_i P_i +_{\mathrm{P}} \bigwedge_i Q_i \\ \bigwedge_i (\top_A \Rightarrow_{\mathrm{P}} Q_i) & = & \top_A \Rightarrow_{\mathrm{P}} \bigwedge_i Q_i \\ \bigwedge_i (P_i \Rightarrow_{\mathrm{P}} Q_i) & \subseteq & \bigwedge_i P_i \Rightarrow_{\mathrm{P}} \bigwedge_i Q_i\end{array}$$

2. *Assume now collections $(S_i \subseteq U \times V)_{i\in I}$ and $(R_i \subseteq X \times Y)_{i\in I}$ of relations.*

$$\begin{array}{rcl}\bigwedge_i (S_i \times_{\mathrm{R}} R_i) & = & \bigwedge_i S_i \times_{\mathrm{R}} \bigwedge_i R_i \\ \bigwedge_i (\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_i) & = & \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \bigwedge_i R_i \\ \bigwedge_i (S_i \Rightarrow_{\mathrm{R}} R_i) & \subseteq & \bigwedge_i S_i \Rightarrow_{\mathrm{R}} \bigwedge_i R_i\end{array}$$

   And under the assumption that I is nonempty

$$\bigwedge_i (S_i +_{\mathrm{R}} R_i) \quad = \quad \bigwedge_i S_i +_{\mathrm{R}} \bigwedge_i R_i$$

The subset relation for the general exponent above, is the best result one can achieve. It is very easy to find examples where the subset relation is strict.

Proof Straightforward after unpacking the definitions. □

Lemma 2.4.10 (Disjunction) *Let A be a set and I an arbitrary index set as before.*

1. *Assume collections of predicates $(P_i \subseteq X)_{i \in I}$ and $(Q_i \subseteq Y)_{i \in I}$.*

$$\begin{array}{rcl} \bigvee_i (P_i \times_{\mathrm{P}} Q_i) & \subseteq & \bigvee_i P_i \times_{\mathrm{P}} \bigvee_i Q_i \\ \bigvee_i (P_i +_{\mathrm{P}} Q_i) & = & \bigvee_i P_i +_{\mathrm{P}} \bigvee_i Q_i \\ \bigvee_i (\top_A \Rightarrow_{\mathrm{P}} Q_i) & \subseteq & \top_A \Rightarrow_{\mathrm{P}} \bigvee_i Q_i \end{array}$$

2. *Assume as before collections of relations $(S_i \subseteq U \times V)_{i \in I}$ and $(R_i \subseteq X \times Y)_{i \in I}$.*

$$\begin{array}{rcl} \bigvee_i (S_i \times_{\mathrm{R}} R_i) & \subseteq & \bigvee_i S_i \times_{\mathrm{R}} \bigvee_i R_i \\ \bigvee_i (S_i +_{\mathrm{R}} R_i) & = & \bigvee_i S_i +_{\mathrm{R}} \bigvee_i R_i \\ \bigvee_i (\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_i) & \subseteq & \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \bigvee_i R_1 \end{array}$$

Again the subset relations in the preceding lemma are the best results that can be achieved in general. The preceding lemma does not contain a statement about the general exponent. The two expressions in question $\bigvee_i (P_i \Rightarrow_{\mathrm{P}} Q_i)$ and $\bigvee_i P_i \Rightarrow_{\mathrm{P}} \bigvee_i Q_i$ $\big(\bigvee_i (R_i \Rightarrow_{\mathrm{R}} S_i)$ and $\bigvee_i R_i \Rightarrow_{\mathrm{R}} \bigvee_i S_i$ for relations$\big)$ are not related at all.

Proof Straightforward after unpacking the definitions. □

Lemma 2.4.11 (Opposite Relation) *For arbitrary relations $S \subseteq U \times V$, $R \subseteq X \times Y$:*

$$\begin{array}{rcl} (S \times_{\mathrm{R}} R)^{\mathrm{op}} & = & S^{\mathrm{op}} \times_{\mathrm{R}} R^{\mathrm{op}} \\ (S +_{\mathrm{R}} R)^{\mathrm{op}} & = & S^{\mathrm{op}} +_{\mathrm{R}} R^{\mathrm{op}} \\ (S \Rightarrow_{\mathrm{R}} R)^{\mathrm{op}} & = & S^{\mathrm{op}} \Rightarrow_{\mathrm{R}} R^{\mathrm{op}} \end{array}$$

Proof Immediate. □

Lemma 2.4.12 (Composition) *Assume four relations $S_1 \subseteq U \times V$, $S_2 \subseteq V \times W$, $R_1 \subseteq X \times Y$, $R_2 \subseteq Y \times Z$ and a set A:*

$$\begin{array}{rcl} (S_1 \times_{\mathrm{R}} R_1) \circ (S_2 \times_{\mathrm{R}} R_2) & = & (S_1 \circ S_2) \times_{\mathrm{R}} (R_1 \circ R_2) \\ (S_1 +_{\mathrm{R}} R_1) \circ (S_2 +_{\mathrm{R}} R_2) & = & (S_1 \circ S_2) +_{\mathrm{R}} (R_1 \circ R_2) \\ (\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_1) \circ (\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_2) & = & \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} (R_1 \circ R_2) \\ (S_1 \Rightarrow_{\mathrm{R}} R_1) \circ (S_2 \Rightarrow_{\mathrm{R}} R_2) & \subseteq & (S_1 \circ S_2) \Rightarrow_{\mathrm{R}} (R_1 \circ R_2) \end{array}$$

Proof Straightforward after unpacking the definitions. The third equation requires the Axiom of Choice. □

There exist examples of relations such that the subset relation in the preceding lemma is strict. The following two lemmas relate the cartesian structure of **Pred** and **Rel**. I need them for the Propositions 2.6.15 and 2.6.17 on page 65f.

Lemma 2.4.13 *Assume two relations $S \subseteq U \times V$ and $R \subseteq X \times Y$ and let A be a set.*

$$\begin{array}{rcl}
\coprod_{\pi_1} (S \times_{\mathrm{R}} R) & = & (\coprod_{\pi_1} S) \times_{\mathrm{P}} (\coprod_{\pi_1} R) \\
\coprod_{\pi_1} (S +_{\mathrm{R}} R) & = & (\coprod_{\pi_1} S) +_{\mathrm{P}} (\coprod_{\pi_1} R) \\
\coprod_{\pi_1} (\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R) & = & \top_A \Rightarrow_{\mathrm{P}} (\coprod_{\pi_1} R) \\
\coprod_{\pi_1} (S \Rightarrow_{\mathrm{R}} R) & \subseteq & (\coprod_{\pi_1} S) \Rightarrow_{\mathrm{P}} (\coprod_{\pi_1} R)
\end{array}$$

It is easy to find an example with $\coprod_{\pi_1} (S \Rightarrow_{\mathrm{R}} R) \not\supseteq (\coprod_{\pi_1} S) \Rightarrow_{\mathrm{P}} (\coprod_{\pi_1} R)$.

Proof Unfold the definitions. The third equation requires the Axiom of Choice. □

Lemma 2.4.14 *Let A, U, V, X, and Y be sets. To avoid confusion denote the first projections as follows*

$$\begin{tikzcd}
X \times Y \arrow[r, "\pi_1"] & X & (A \Rightarrow X) \times (A \Rightarrow Y) \arrow[r, "\pi_4"] & A \Rightarrow X \\
U \times V \arrow[r, "\pi_2"] & U & (U \Rightarrow X) \times (V \Rightarrow Y) \arrow[r, "\pi_5"] & U \Rightarrow X \\
(U \times X) \times (V \times Y) \arrow[r, "\pi_3"] & U \times X & (U + X) \times (V + Y) \arrow[r, "\pi_6"] & U + X
\end{tikzcd}$$

Let $P \subseteq X$ and $Q \subseteq U$ be predicates, then

$$\begin{array}{rcl}
\pi_3^* (Q \times_{\mathrm{P}} P) & = & (\pi_2^* Q) \times_{\mathrm{R}} (\pi_1^* P) \\
\pi_4^* (\top_A \Rightarrow_{\mathrm{P}} P) & = & \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} (\pi_1^* P) \\
\pi_5^* (Q \Rightarrow_{\mathrm{P}} P) & \subseteq & (\pi_2^* Q) \Rightarrow_{\mathrm{R}} (\pi_1^* P)
\end{array}$$

For the coproduct assume additionally two relations $R \subseteq X \times Y$ and $S \subseteq U \times V$

$$(S +_{\mathrm{R}} R) \wedge \pi_6^* (Q +_{\mathrm{P}} P) \quad = \quad (S +_{\mathrm{R}} R) \wedge (\pi_2^* Q) +_{\mathrm{R}} (\pi_1^* P)$$

Under the assumption that V is nonempty:

$$\pi_5^* (Q \Rightarrow_{\mathrm{P}} P) \quad = \quad (\pi_2^* Q) \Rightarrow_{\mathrm{R}} (\pi_1^* P)$$

Proof By unfolding the definitions. □

Next I consider fibredness and cofibredness of the bicartesian structure of **Pred** and **Rel** over their counterparts in **Set**. The next result is well known and can be found, for instance in (Hensel, 1999).

Lemma 2.4.15 (Fibredness)

1. *The product, coproduct, and the exponent of* **Pred** *are fibred over the corresponding structure in* **Set**.

2. The same applies to the product, coproduct, and the exponent of **Rel***.*

Proof This Lemma has also been proved completely in PVS. For illustration I show that the exponent in **Rel** is fibred. We have to establish that for arbitrary relations $(S \subseteq U \times V)$ and $(R \subseteq X \times Y)$ and functions $u : U \longrightarrow U', v : V \longrightarrow V', f : X' \longrightarrow X$, and $g : Y' \longrightarrow Y$ it holds that

$$\big(\coprod_{(u\times v)} S\big) \Rightarrow_{\mathrm{R}} \big((f \times g)^* R\big) \quad = \quad ((u \Rightarrow f) \times (v \Rightarrow g))^* \ (S \Rightarrow_{\mathrm{R}} R)$$

(compare Section 2.3, pages 31ff.) This is done by

$$\begin{aligned}
\big(\coprod_{(u\times v)} S\big) \Rightarrow_{\mathrm{R}} \big((f \times g)^* R\big) \quad &\subseteq \quad (U' \Rightarrow X') \times (V' \Rightarrow Y') \\
= \quad & \big\{(a,b) \mid \forall p' : U', q' : V' . \\
& \qquad \big(\exists p : U, q : V . p' = u(p) \ \text{ and } \ q' = v(q) \ \text{ and } \ S(p,q)\big) \\
& \qquad\quad \text{implies} \ \ R(\, f(a(p')), g(b(q'))\,)\big\} \\
= \quad & \big\{(a,b) \mid \forall p : U, q : V . S(p,q) \ \text{ implies } \ R\big(f(a(u(p))), g(b(v(q)))\big)\big\} \\
= \quad & ((u \Rightarrow f) \times (v \Rightarrow g))^* \, (S \Rightarrow_{\mathrm{R}} R) \qquad \square
\end{aligned}$$

It is a new observation, that the bicartesian structure of both **Pred** and **Rel** is not cofibred. Although product and coproduct are cofibred, for the exponent cofibredness fails.

Example 2.4.16 I show an example in which the cofibredness condition for the exponent in **Pred** fails. That is, I construct predicates $P \subseteq X$, $Q \subseteq Y$ and functions $f : U \longrightarrow X$ and $g : Y \longrightarrow V$ such that in the fibre over $U \Rightarrow V$

$$(f^* P) \Rightarrow_{\mathrm{P}} (\coprod_g Q) \quad \not\cong \quad \coprod_{f \Rightarrow g} (P \Rightarrow_{\mathrm{P}} Q) \qquad (*)$$

Take as four concrete sets

$$U \overset{\text{def}}{=} \{u_0, u_1\} \qquad\qquad V \overset{\text{def}}{=} \{v_0, v_1\}$$
$$X \overset{\text{def}}{=} \{x_0, x_1\} \qquad\qquad Y \overset{\text{def}}{=} \{y_0, y_1\}$$

Let $P \overset{\text{def}}{=} \{x_0\}$ and $Q \overset{\text{def}}{=} \{y_0\}$ be the predicates over X and Y respectively. Define the functions

$$\begin{aligned}
f \ &: \ U \longrightarrow X \ \overset{\text{def}}{=} \ \lambda u : U . \, \mathsf{if} \ u = u_o \ \mathsf{then} \ x_0 \ \mathsf{else} \ x_1 \ \mathsf{endif} \\
g \ &: \ Y \longrightarrow V \ \overset{\text{def}}{=} \ \lambda y : Y . \, v_1
\end{aligned}$$

There are exactly four functions $U \longrightarrow V$, namely

$$h_1 \overset{\text{def}}{=} \lambda u : U . \, v_0 \qquad h_3 \overset{\text{def}}{=} \lambda u : U . \, \mathsf{if} \ u = u_0 \ \mathsf{then} \ v_0 \ \mathsf{else} \ v_1 \ \mathsf{endif}$$
$$h_2 \overset{\text{def}}{=} \lambda u : U . \, v_1 \qquad h_4 \overset{\text{def}}{=} \lambda u : U . \, \mathsf{if} \ u = u_0 \ \mathsf{then} \ v_1 \ \mathsf{else} \ v_0 \ \mathsf{endif}$$

Computing now both sides of $(*)$ yields

$$(f^* P) \Rightarrow_{\mathrm{P}} (\textstyle\coprod_g Q) \quad = \quad \{h_2, h_4\} \quad \not\supseteq \quad \{h_2\} \quad = \quad \textstyle\coprod_{f \Rightarrow g} (P \Rightarrow_{\mathrm{P}} Q) \qquad \blacksquare$$

The preceding example shows that in general the exponent is not cofibred in **Pred**. However, cofibredness holds in the restricted case of constant arguments. This allows me to derive a cofibredness results for polynomial functors (in Lemma 3.4.4 on page 95 and Lemma 3.4.7 on page 98 below).

Lemma 2.4.17 (Cofibredness)

1. *The product and the coproduct both of* **Pred** *and* **Rel** *are cofibred over the corresponding structure in* **Set**.
2. *Let A be an arbitrary set, $(P \subseteq X)$ be a predicate and f a function $X \longrightarrow X'$. Then for the exponent in* **Pred** *it holds that*

$$\top_A \Rightarrow_{\mathrm{P}} \textstyle\coprod_f P \quad = \quad \textstyle\coprod_{\mathrm{id}_A \Rightarrow f} (\top_A \Rightarrow_{\mathrm{P}} P)$$

3. *Let A again be a set and assume further a relation $R \subseteq X \times Y$ and two functions $f : X \longrightarrow X'$ and $g : Y \longrightarrow Y'$. Then*

$$\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \textstyle\coprod_{f \times g} R \quad = \quad \textstyle\coprod_{(\mathrm{id}_A \Rightarrow f) \times (\mathrm{id}_A \Rightarrow g)} \big(\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R\big)$$

Proof This Lemma has been proved completely in PVS. Item (3) is the most difficult part. I show its proof here. For the left hand side I compute

$$\begin{aligned} \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \textstyle\coprod_{f \times g} R \quad &\subseteq \quad (A \Rightarrow X') \times (A \Rightarrow Y') \\ = \quad &\big\{(h', k') \mid \forall a : A \,.\, \exists x : X, y : Y \,.\, R(x, y) \text{ and} \\ &\quad h'(a) = f(x) \text{ and } k'(a) = g(y)\big\} \end{aligned} \qquad (*)$$

And for the right hand side

$$\begin{aligned} &\textstyle\coprod_{(\mathrm{id}_A \Rightarrow f) \times (\mathrm{id}_A \Rightarrow g)} \big(\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R\big) \\ &= \quad \big\{(h', k') \mid \exists h : A \longrightarrow X, k : A \longrightarrow Y \,. \\ &\qquad \forall a : A \,.\, R(h(a), k(a)) \text{ and } h' = f \circ h \text{ and } k' = g \circ k\big\} \end{aligned} \qquad (\dagger)$$

There is an obvious inclusion from $(\dagger)$ to $(*)$. For the other direction it is necessary to construct suitable functions h and k using the Axiom of Choice. □

2.4.4. PVS Formalisation

As already mentioned before all lemmas and examples of the preceding subsection have been formalised and proved in PVS. This applies also to some more material from Chapter 3 (including all examples). The PVS sources are available at URL http://wwwtcs.inf.tu-dresden.de/~tews/PhD/. They have been developed and checked with PVS version 2.4 patch level 1.

A similar formalisation of concepts of fibred category theory has also been done by Hensel. In Chapter 3 of (Hensel, 1999) he proves that the logic of PVS fulfils the assumptions of his main theorem about the validity of the induction and the coinduction principle. In particular he shows that PVS admits comprehension and quotients. In comparison with my formalisation the main difference is a conceptual one: While Hensel uses PVS as an example of a general theory, I use PVS to prove (to that extend that is feasible) the main results of Chapter 3. So for me PVS is a tool that helped in the development and in the validation of my main results about different generalisations of polynomial functors.

Despite of this conceptual difference the setup of both formalisation is very much the same. Therefore I only sketch here the main ideas of my formalisation. The Appendix A illustrates my formalisation of the fibrations of predicates and relations with some excerpts from the PVS source code. The appendix also contains a table that relates the lemmas and propositions of this thesis with PVS statements in the formalisation. A more detailed description is on the world wide web at the URL given above.

PVS implements a simply typed higher-order logic with predicate subtypes, dependent types, and a few other extensions. The PVS sources are organised in *theories*. Theories can have type parameters and value parameters. The type parameters add a form of polymorphism to the logic of PVS. The value parameters give an additional level of dependent typing.

The main idea of the formalisation is the following: The PVS universe of types forms the collection of objects of a base category $\mathbb{B}_{\text{PVS}}$. The set of functions between two types gives the morphisms between two objects. Statements and constructions that are parametric in n objects are placed in a theory with n different type parameters. For instance the construction of the product $f \times g$ of two morphisms is parametric in four objects (domains and codomains of both f and g) so it is placed into a theory with four type parameters. This way the construction and the statements that are proved in PVS apply to all possible instantiations, hence to all objects of $\mathbb{B}_{\text{PVS}}$.

The total categories **Pred** and **Rel** are formalised as a collection of fibres over $\mathbb{B}_{\text{PVS}}$. A fibre over an object X (respectively $X \times Y$ for **Rel**) is modelled by the type of predicates over X (respectively $X \times Y$). In higher-order logic the type of predicates over a type τ is conveniently modelled with the function type $\tau \Rightarrow \mathsf{bool}$. This way constructions in the total category map quite naturally to manipulations of predicates in PVS. For instance the substitution functor $(-)^*$ is a construction that is parametric in two objects (the domain and the codomain of the morphism in the base). Therefore it is placed in a PVS theory with two type parameters, say X and Y. For two fixed objects the substitution functor is a functional (i.e., a function on function spaces) that maps a function $X \longrightarrow Y$ and a function $Y \longrightarrow \mathsf{bool}$ (the predicate over Y) to a function $X \longrightarrow \mathsf{bool}$ (a predicate over X).

This approach of formalising the fibrations of predicates and relations in PVS has the advantage that the mapping of abstract categorical properties into the logic of PVS is rather straightforward. For instance the exponent of objects is mapped to the built-in function type. The resulting proof obligations are such that the strategies and decision procedures that are built-in into PVS work very efficiently. Many theorems can be proved with grind, the most powerful proof strategy of PVS. The disadvantage of mapping objects directly to types is that one cannot represent functors in the formalisation, because PVS does not support mappings from types to types.

Any particular functor can be represented as a type expressions in a parametrised theory. The theory parameters play then the role of the arguments of the functor. (This idea can be traced back to (Hensel et al., 1998).) However one cannot formalise a statement about a set (or a class) of functors, because this would require quantification over parametric type expressions, which is not available in PVS.

The main theorems of Chapter 3 involve a quantification over a class of functors, for instance over the class of *extended polynomial functors* introduced in Section 3.4. The different generalisations of polynomial functors that are of interest in the present thesis are all finitely generated by the bicartesian structure of the base category. Many theorems about these functors are proved by induction on their structure. The induction steps look typically like: Assume that a property P holds for both functors F_1 and F_2, show that P holds also for the functor $F(X) = F_1(X) \times F_2(X)$. Because the bicartesian structure is formalised in PVS, the induction step can be mimicked in PVS in the following way: Assume that the type parameters T_1 and T_2 both have a property P, show that the property holds also for the product type $T_1 \times T_2$.

So the bottom line is the following: The main theorems of Chapter 3 have not been formalised in PVS. However, for those theorems that are proved by induction on the structure of the involved functor, all induction steps have been formalised and proved in PVS.

An alternative approach would be to axiomatise a bicartesian closed category as a structure over two types, one for the objects and one for the morphisms. Then, a functor could be represented as a pair of functions. However, I suspect that, under this approach, the translation from abstract properties into PVS would be much more complicated. Also the built-in decision procedures of PVS would not work that well and proofs would be more complicated. Another drawback is that all proofs would depend on the consistency of the axiomatisation, which could not be established within PVS. In contrast, my approach does not use any axioms.

The size of my formalisation of the fibrations of predicates and relations is considerable. It contains about 800 statements (theorems) in the logic of PVS. They are spread over more than 200KB PVS source code in 37 files. The proof scripts (that allow one to rerun and to check all the proves) contain altogether more than 10.000 proof commands. I chose PVS for the formalisation because at that time PVS was the only theorem prover I was acquainted with. My experience with PVS is mixed: On the one hand recent versions of PVS work quite efficient on such large specifications. One can easily change or add declarations and theorems, PVS keeps track of what has to get invalidated. Another impressive feature of PVS are the decision procedures. Often the high-level proof strategies of PVS were able to prove a theorem, without that I actually understood the details of the proof. In order to understand them I proved some theorems a second time using only simple proof commands.

On the other hand, considered as a software system, PVS has considerable quality problems. While working on the formalisation I submitted more than 40 bug reports, see http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/ (so on average every 200 lines of source code triggered a bug). The Fiasco case study that is described in Section 4.10 (on page 235) was originally done with PVS version 2.2. Since then, more than three years later, version 2.3 and 2.4 (and several patches) have been released. However, all of these successor versions contain a bug[4] that makes it impossible to port the case study to any of the newer versions.

(For a comparison of PVS and ISABELLE/HOL see (Griffioen and Huisman, 1998) and Section 8.2 in (Huisman, 2001).)

2.5. Algebras

The main emphasis of this thesis is on *co*algebras and on *co*algebraic specification. Algebras are included here mainly for two reasons. First, it might be easier for the uninitiated reader to understand coalgebras in contrast to algebras. Second, the specification language CCSL (described in Chapter 4) allows the user to specify algebras. So to make this thesis self contained algebras are formally defined in this section. To make the duality between algebras and coalgebras more obvious (and therefore to make it easier to understand the following section on coalgebras) I use the categorical definition for algebras. Algebras and algebraic specification is treated in detail in (Wirsing, 1990).

Algebras are widely accepted as the right formalism to specify finitely generated data structures such as lists or trees. I take the data type $\mathsf{List}[A]$ of finite list over A as the running example of this section. Two operations suffice to generate all elements of $\mathsf{List}[A]$. The first operation is $\mathsf{nil} : \mathbf{1} \longrightarrow \mathsf{List}[A]$, the empty list. The second operation is $\mathsf{cons} : A \times \mathsf{List}[A] \longrightarrow \mathsf{List}[A]$. Additional operations like concatenation or reversal of lists can be defined by induction (or primitive recursion), so I leave them aside for a while.

Both operations nil and cons have as codomain the type $\mathsf{List}[A]$ that I am about to define. This is a fundamental observation: An algebraic signature consists of (a finite set

[4] See problem report #516 in the PVS bug tracking system.

of) operations of the form

$$\boxed{\cdots\ \ \mathsf{Carrier}\ \ \cdots} \longrightarrow \mathsf{Carrier} \tag{$*$}$$

Operations of this form are called *constructors* or *algebras*. They describe the primitive operations that are available to build elements of $\mathsf{Carrier}$, the carrier set of the new data type.

All constructors in a signature can be combined into one algebra using the coproduct. In the list example the two constructors nil and cons are equivalent to one algebra $\alpha_{\mathsf{List}} : \mathbf{1} + (A \times \mathsf{List}[A]) \longrightarrow \mathsf{List}[A]$. From the combined list algebra α_{List} one gets the operations nil and cons by precomposing injections, for instance $\alpha_{\mathsf{List}} \circ \kappa_0 = \mathsf{nil}$. Observe that all information of the signature is contained in the domain type of the combined algebra. Thus one can describe the signature of a data type by an endofunctor that maps the carrier set to the domain of the algebra.

For a formal definition of the term algebra consider an arbitrary category $\mathbb{C}$ and an endofunctor $T : \mathbb{C} \longrightarrow \mathbb{C}$. A *$T$–algebra* is a morphism $T(X) \longrightarrow X$ in $\mathbb{C}$. The functor T plays the role of signatures in traditional universal algebra. The categorical definition of algebra is more general. It applies to arbitrary categories and not just to **Set**. Further even on the category **Set**, the notion of an endofunctor is more general than that of a (single sorted) algebraic signature.

The functor that describes the list signature is $T_{\mathsf{List}}(X) = \mathbf{1} + A \times X$, where A is the set of possible elements of the list. In the category **Set** there are a lot of sets M that allow one to define a list algebra (i.e., a function $T_{\mathsf{List}}(M) \longrightarrow M$). Most of them have nothing in common with finite lists over A. Therefore we need some means to restrict our attention to the *interesting* list algebras. This can be done in two different ways, either internally by stating properties about the set M, or externally by relating M to other list algebras.

For an internal characterisation we state that $\mathsf{List}[A]$ is the carrier set of a list algebra with the following two properties: First, all elements of $\mathsf{List}[A]$ can be finitely constructed by using only the list algebra α_{List}, that is, with the two operations nil and cons. Second, any two such finite constructions yield different elements in $\mathsf{List}[A]$. It is easy to see that any list algebra that fulfils both points is isomorphic to the set of finite list over A.

For the external characterisation we need the notion of a *structure preserving map*, often called List *homomorphism*, or in categorical terms List *algebra morphism*. Assume we have two list algebras, that is, two sets M and N with associated operations $\mathsf{nil}_M, \mathsf{cons}_M, \mathsf{nil}_N$, and cons_N. A function $f : M \longrightarrow N$ is a list algebra homomorphism if we have both that $\mathsf{nil}_N = f(\mathsf{nil}_M)$ and $\mathsf{cons}_N(a, f(m)) = f(\mathsf{cons}_M(a, m))$ for all $m \in M$. The external characterisation is now as follows: The set $\mathsf{List}[A]$ is the carrier set of a list algebra α_{List} for which there is precisely one list algebra homomorphism to any list algebra.

Both the internal and the external characterisation are equivalent for arbitrary algebraic signatures. In this thesis I prefer to work with the external characterisation because it can be expressed categorically as follows.

An arrow $f : X \longrightarrow Y$ of $\mathbb{C}$ is a *T–algebra morphism* between two T–algebras $\alpha : T(X) \longrightarrow X$ and $\beta : T(Y) \longrightarrow Y$ if the following diagram commutes.

$$\begin{tikzcd}
T(X) \arrow[r, "\alpha"] \arrow[d, "T(f)"'] & X \arrow[d, "f"] \\
T(Y) \arrow[r, "\beta"] & Y
\end{tikzcd} \tag{2.5}$$

For a given endofunctor T, the T–algebras and their morphisms form a category, denoted by $\mathcal{A}lg(T)$. The identities in $\mathcal{A}lg(T)$ are identity functions and the composition of two T–algebra morphisms is given by the composition of the underlying functions. The initial T–algebra is the initial object in $\mathcal{A}lg(T)$. If the initial algebra for a functor exists, then it is an isomorphism.

In algebraic specification one considers algebraic signatures together with a set of axioms that describe additional properties of the constructors. In this thesis I would like to restrict the attention to *abstract data type specification*, that is algebraic specifications without Axioms. I use the term *abstract data type* to refer to structures that are constructed as initial algebra.

Algebraic specification is a powerful technique. It allows one to define new abstract data types up to isomorphism without referring to any internal representation. Further the *induction principle* holds for abstract data types. The induction principle can be exploited in two different ways: First to define functions via primitive recursion and second to prove properties for all elements of an abstract data type. Let me demonstrate how induction as definition and as proof principle is connected to initiality.

Assume we want to define the length function $\mathsf{List}[A] \longrightarrow \mathbb{N}$ inductively. All we have to do for that is to define an appropriate List algebra structure on $\mathbb{N}$. For the length function it is given by

$$\begin{aligned}
\mathsf{nil}_{\mathsf{length}} &: \mathbf{1} \longrightarrow \mathbb{N} \\
&= 0 \\
\mathsf{cons}_{\mathsf{length}} &: A \times \mathbb{N} \longrightarrow \mathbb{N} \\
&= \lambda a : A, n : \mathbb{N} \,.\, n + 1
\end{aligned}$$

The initiality of the abstract data type of lists provides us with a List–algebra morphism $\mathsf{length} : \mathsf{List}[A] \longrightarrow \mathbb{N}$. The commutation property of Diagram 2.5 amounts for the algebra morphism length to the following two familiar equations.

$$\begin{aligned}
\mathsf{length}(\mathsf{nil}) &= 0 \\
\mathsf{length}(\mathsf{cons}(a, l)) &= \mathsf{length}(l) + 1
\end{aligned}$$

In set theory the induction proof principle comes out of the fact that the initial algebra has no proper subalgebras. A formulation of induction in terms of fibrations has the advantage, that it makes induction available in the syntactic language of a logic. The general theory stems from (Hermida and Jacobs, 1998), here I show how this works in case of the predicate fibration $\begin{smallmatrix}\mathbf{Pred}\\ \downarrow\\ \mathbf{Set}\end{smallmatrix}$.

Consider the functor $\mathrm{Pred}(T_{\mathsf{List}}) : \mathbf{Pred}_X \longrightarrow \mathbf{Pred}_{T_{\mathsf{List}}(X)}$ between the fibres of **Pred** defined as

$$\mathrm{Pred}(T_{\mathsf{List}})(P \subseteq X) \qquad = \qquad \top_{\mathbf{1}} +_{\mathrm{P}} (\top_A \times_{\mathrm{P}} P)$$

(The functor $\mathrm{Pred}(T_{\mathsf{List}})$ is called the predicate lifting of T_{List}, it will be defined systematically in Definition 2.6.3 in the next section.) Algebras for the functor $\mathrm{Pred}(T_{\mathsf{List}})$ are morphisms $(\mathrm{Pred}(T_{\mathsf{List}})(P) \subseteq T_{\mathsf{List}}(X)) \longrightarrow (P \subseteq X)$ in **Pred** that consist of a T_{List}–algebra on X together with two implications $\top \supset P(\mathsf{nil}_X)$ and $P(x) \supset P(\mathsf{cons}_X(a, x))$. Hermida and Jacobs prove in (Hermida and Jacobs, 1998) that there is a functor $\mathcal{A}lg(T_{\mathsf{List}}) \longrightarrow \mathcal{A}lg(\mathrm{Pred}(T_{\mathsf{List}}))$ that preserves initial objects and sends the initial T_{List}–algebra to an algebra in **Pred** with carrier $(\top_{\mathsf{List}[A]} \subseteq \mathsf{List}[A])$. This shows that the induction proof principle for lists is valid in $\begin{smallmatrix}\mathbf{Pred}\\ \downarrow U\\ \mathbf{Set}\end{smallmatrix}$: Assume we have a predicate $(P \subseteq \mathsf{List}[A])$ and can prove the two implications $\top \supset P(\mathsf{nil})$ and $P(x) \supset P(\mathsf{cons}(a, x))$. This means we have a $\mathrm{Pred}(T_{\mathsf{List}})$ algebra with carrier $(P \subseteq \mathsf{List}[A])$ in Pred. Because the $\mathrm{Pred}(T_{\mathsf{List}})$ algebra on $\top_{\mathsf{List}[A]}$ is initial we have also a (vertical) morphism $\top_{\mathsf{List}[A]} \longrightarrow P$. This last morphism corresponds to a proof that P holds for all lists.

2.6. Coalgebras

In this section I introduce coalgebras and explain how coalgebras can be used to describe processes or classes of object-oriented programming languages. Coalgebras and coalgebraic specification are in the centre of this thesis. Chapter 3 proposes an extension of the classical notion of coalgebras that makes it possible to model classes with methods of arbitrary types, including binary methods. To judge the results obtained there it is necessary to have an overview about the properties of the classical notion of coalgebra. This section will therefore review many known results about coalgebras. They have been taken from (Rutten, 2000; Jacobs, 1997b), but see also (Gumm, 1999). A general introduction into coalgebras and the related notions of coinduction and bisimulation is (Jacobs and Rutten, 1997).

With few exceptions all the results of this section are proved for a more general notion of coalgebra in Chapter 3. Therefore the proofs in this section have mainly informative character.

Algebras are good for describing constructions, where the user wants to reason about the internal structure. Coalgebras are good for describing *observations* where the internal structure is hidden. Consider the set of possibly infinite sequences $\mathsf{Seq}[A]$ of elements from A. It comes together with the following operation:

$$\mathsf{next} : \ \mathsf{Seq}[A] \longrightarrow (A \times \mathsf{Seq}[A]) + \mathbf{1} \tag{$*$}$$

(Recall that in **Set** we have $\mathbf{1} = \{*\}$ and that the coproduct is given by the disjoint union.) If we have a sequence $s \in \mathsf{Seq}[A]$ then either $\mathsf{next}(s) = \bot = \kappa_2 *$ or $\mathsf{next}(s) = \kappa_1(a, s')$ for some a and s'. The first case means that s is empty and contains no more elements. The second case means that the first element of s is a and s' is the sequence that contains the remaining elements.

Note that next has a structured codomain, so it does not fit into an algebraic signature. A function of this shape

$$\mathsf{Self} \longrightarrow \boxed{\ \cdots \ \ \mathsf{Self} \ \ \cdots\ }$$

is called a coalgebra. The set Self is the *state space* of the coalgebra (sometimes also called the carrier set). Note that with a coalgebra one can only make observations, one cannot inspect the internal structure of an element of the state space. Note also, that the next in $(*)$ can be seen as a partial function. It fails for those sequences that are empty.

Two coalgebras on the same carrier can be combined into one. If c_1 and c_2 are coalgebras on the carrier X then their pairing $\langle c_1, c_2 \rangle$ is also a coalgebra on X. One obtains the original coalgebras by postcomposing a projection, for instance $\pi_1 \circ \langle c_1, c_2 \rangle = c_1$. So in theoretical investigations it is enough to consider just one coalgebra.

The structured codomain of the coalgebra, depicted as $\boxed{\ \cdots \ \ \mathsf{Self} \ \ \cdots\ }$ above, corresponds to a (coalgebraic) signature. It describes possible observations that can be made about an element $x \in \mathsf{Self}$ with one application of the coalgebra and how to obtain successor states. Informally I call this the *interface type* of the coalgebra. Formally the interface type is described by a functor. In the sequence example the interface type is given by the functor $T_{\mathsf{Seq}}(X) = (A \times X) + 1$. For a sequence s the following observations are possible: Either $\mathsf{next}\, s = \kappa_1(\cdots)$ or $\mathsf{next}\, s = \kappa_2 *$. In the first case we can destruct the result further and obtain (via the first projection) an additional observation in A and (via the second projection) a successor state.

The *observable behaviour* of $x \in \mathsf{Self}$ is the tree of observations that results from the application of the coalgebra to x and successively to all its successor states. The observable behaviour of an element $s \in \mathsf{Seq}[A]$ is a finite or infinite sequence in A (which should not come as a surprise now).

Coalgebras and their use in specification are the subject of this thesis. Let me give two possible interpretations for a coalgebra $\alpha : X \longrightarrow T(X)$ of interface type $T(X)$. First, you can think of X as a set of possible states of an automaton.The interface type $T(X)$ gives the number and the type of the observations that are possible on these automata. The coalgebra α corresponds to a transition function that, for a given state $x \in X$, delivers all possible observations for x and all possible successor states of x *at once*.

A second analogy that gives significance to coalgebras is that of object-oriented programming. A class interface (i.e., the number and type of the methods and instance variables) gives rise to a functor T. A coalgebra $X \longrightarrow T(X)$ corresponds then to a class, that is, to a fixed implementation of all the methods. The set X is the set of all possible (states of) objects of that class. Important phenomena of object-oriented programming like inheritance, aggregation through components, and dynamic binding can be modelled with coalgebras, see (Jacobs, 1996b; Hensel et al., 1998). It is this second analogy, that motivates the present thesis.

Definition 2.6.1 (Category of Coalgebras) Let $T : \mathbb{C} \longrightarrow \mathbb{C}$ be an endofunctor on a category $\mathbb{C}$.

- A *T–coalgebra* is a morphism $X \longrightarrow T(X)$ in $\mathbb{C}$.
- A *T–coalgebra morphism* between two T–coalgebras $\alpha : X \longrightarrow T(X)$ and $\beta : Y \longrightarrow T(Y)$ is a morphism $f : X \longrightarrow Y$ in $\mathbb{C}$ such that the following diagram commutes.

$$\begin{tikzcd}
X \arrow[r, "\alpha"] \arrow[d, "f"'] & T(X) \arrow[d, "T(f)"] \\
Y \arrow[r, "\beta"] & T(Y)
\end{tikzcd} \tag{2.6}$$

- Identity morphisms are the identities from $\mathbb{C}$. The composition of two T–coalgebra morphisms is their composition in $\mathbb{C}$.

This forms, for any endofunctor T, the category $CoAlg(T)$ of T–coalgebras.

In the sequence example coalgebra morphisms are as follows. Let $\alpha : X \longrightarrow T_{\mathsf{Seq}}(X)$ and $\beta : Y \longrightarrow T_{\mathsf{Seq}}(Y)$ be two sequence coalgebras. A function $f : X \longrightarrow Y$ is a sequence coalgebra morphism (i.e., a structure preserving map between sequences) if the following holds for all $x \in X$

$$\beta(f(x)) \quad = \quad \begin{cases} \kappa_1(a, f(x')) & \text{if } \alpha(x) = \kappa_1(a, x') \\ \kappa_2 * & \text{if } \alpha(x) = \kappa_2 * \end{cases}$$

Note that in case f is a sequence morphism, both x and $f(x)$ generate the same observable behaviour.

If T is an endofunctor on **Set** one says that a coalgebra $d : Y \longrightarrow T(Y)$ is a subcoalgebra of $c : X \longrightarrow T(X)$ if $Y \subseteq X$ and if the inclusion $\iota : Y \hookrightarrow X$ is a coalgebra morphism $d \xrightarrow{\iota} c$.

For natural transformations there is the following result.

Proposition 2.6.2 *A natural transformation $\eta : T \Longrightarrow F$ between two endofunctors gives rise to a functor $CoAlg(T) \longrightarrow CoAlg(F)$. Its object part maps a T–coalgebra $c : X \longrightarrow T(X)$ to $\eta_X \circ c : X \longrightarrow F(X)$. Its morphism part is the identity.*

Proof The main proof obligation is to show that T-coalgebra morphisms are mapped to F coalgebra morphisms. This is proved by the following diagram:

$$\begin{tikzcd}
X \arrow[d, "f"] & X \arrow[r, "c"] \arrow[d, "f"] & T(X) \arrow[r, "\eta_X"] \arrow[d, "T(f)"] & F(X) \arrow[d, "F(f)"] \\
Y & Y \arrow[r, "d"] & T(Y) \arrow[r, "\eta_Y"] & F(Y)
\end{tikzcd}$$

□

2.6.1. Polynomial Functors

Functors play the role of signatures in that they describe what observations are possible on a given coalgebra. Similar to restricting the types that may occur in a signature one restricts the class of functors that are considered. The simplest class considered in the present thesis are the *polynomial functors.*

Assume a bicartesian closed category $\mathbb{C}$. An endofunctor $\mathbb{C} \longrightarrow \mathbb{C}$ is called a *polynomial functor* if it is defined as one of the following cases

$$F(X) \quad = \quad \begin{cases} X \\ A \\ F_1(X) \times F_2(X) \\ F_1(X) + F_2(X) \\ A \Rightarrow F_1(X) \end{cases}$$

where A is an arbitrary (constant) object of $\mathbb{C}$ and F_1 and F_2 are previously defined polynomial functors. The morphism part is defined in the obvious way:

$$F(f) \quad = \quad \begin{cases} f \\ \mathrm{id}_A \\ F_1(f) \times F_2(f) \\ F_1(f) + F_2(f) \\ \mathrm{id}_A \Rightarrow F_1(f) \end{cases} \qquad \text{if } F(X) \; = \; \begin{cases} X \\ A \\ F_1(X) \times F_2(X) \\ F_1(X) + F_2(X) \\ A \Rightarrow F_1(X) \end{cases}$$

There are small but subtle differences in the use of the term polynomial functor in the literature. The present definition coincides with (Jacobs, 1997b) and (Rutten, 2000). Both (Hermida and Jacobs, 1998) and (Poll and Zwanenburg, 2001) do not allow the constant exponent $A \Rightarrow F_1(X)$ in their notion of polynomial functor.

Polynomial functors can model many types of observations that occur in practice. However there are the following restrictions: First, with polynomial functors one cannot model observations in an abstract data type. The function

$$\mathsf{Self} \longrightarrow \mathsf{List}[A \times \mathsf{Self}] \tag{†}$$

cannot be transformed into a coalgebra for a polynomial functor. In his thesis (Hensel, 1999) defines the class of *data functors* and investigates their properties. Data functors allow the iterated use of least and greatest fixed-point constructions, so the interface type in (†) can be modelled by a data functor.

Second, polynomial functors cannot model nondeterminism. To overcome this restriction many papers allow the finite powerset construction in the interface functors, see for instance (Rutten, 2000; Jacobs, 1996b; Jacobs, 2000; Gumm, 1999). I do not consider the finite powerset functor in this thesis.

Third, the exponent is allowed in polynomial functors, but only if its domain is a constant. Consider a function

$$\mathsf{Self} \times A \longrightarrow (\mathsf{Self} \times B) + \mathbf{1}$$

it can be considered as a coalgebra, because after currying it is equivalent to

$$\mathsf{Self} \longrightarrow A \Rightarrow \big((\mathsf{Self} \times B) + \mathbf{1}\big)$$

Consider now

$$\mathsf{Self} \times \mathsf{Self} \longrightarrow (\mathsf{Self} \times B) + \mathbf{1} \tag{‡}$$

with currying one obtains

$$\mathsf{Self} \longrightarrow \mathsf{Self} \Rightarrow \big((\mathsf{Self} \times B) + \mathbf{1}\big)$$

but here the domain of ⇒ is not a constant, so the latter interface type cannot be represented with a polynomial functor. In object-oriented programming operations like (‡) are called *binary methods*. Binary methods are the subject of Chapter 3. There I consider interface functors and coalgebras that can model arbitrary polynomial method types, including binary methods. The specification language CCSL (described in Chapter 4) admits both method types (†) and (‡). To have nondeterminism in CCSL one has to declare a type constructor for the powerset functor as a ground signature extension.

2.6.2. Bisimulations and Invariants

A bisimulation is a relation on the state space of a coalgebra that relates states that cannot be distinguished by their observable behaviour. An invariant is a predicate on the state space that, once it holds for a state x, remains valid for all states that can be reached from x. Let me give an example before I proceed with the technicalities.

For a sequence coalgebra $\alpha : X \longrightarrow (A \times X) + \mathbf{1}$ and a state $x \in X$ it is possible to distinguish if x is empty. If x is nonempty one can observe the first element in x. A relation $R \subseteq X \times X$ is a bisimulation for α if it relates only elements that cannot be distinguished by observations. Formally, R must fulfil:

$$x\,R\,y \qquad \text{implies} \qquad \begin{cases} \alpha\,x = \kappa_1(a, x') \wedge \alpha\,y = \kappa_1(b, y') \wedge a = b \wedge x' R y' & \text{or} \\ \alpha\,x = \alpha\,y = \kappa_2\,* \end{cases}$$

for all $x, y \in X$.

Bisimulations date back to (Milner, 1989) where he used them to relate processes with the same behaviour in the process calculus CCS. Bisimulations play also an important role in judging the expressiveness of modal logics, see for instance (Stirling, 1992). For coalgebras one needs a definition of bisimulation which is parametric in the functor that describes the interface type. There are two traditions: Following Aczel and Mendler in (Aczel and Mendler, 1989), a bisimulation is the state space of a coalgebra which makes a certain diagram (of coalgebra morphisms) commute. Rutten, Hennicker, and Kurz follow this approach in (Rutten, 2000) and (Hennicker and Kurz, 1999), respectively. In this thesis I will call this the Aczel/Mendler approach and when necessary I will use the term *Aczel/Mendler bisimulation*.

The second tradition stems from Hermida and Jacobs. In (Hermida and Jacobs, 1998) they define a special operation —called *relation lifting*— for polynomial functors. Relation lifting is then used to define bisimulations. This approach is used for instance in (Hensel, 1999; Hensel et al., 1998; Jacobs, 1997b). When necessary I use the term *Hermida/Jacobs bisimulation* to avoid confusion.

Similar to bisimulations invariants can be either defined as subcoalgebras or via *predicate lifting*. I use the terms *Aczel/Mendler invariant* and *Hermida/Jacobs invariant* to distinguish both definitions.

Both the Aczel/Mendler approach and the Hermida/Jacobs approach to define bisimulations and invariants have their advantages. Without discussing this in full detail, I only note, that the Aczel/Mendler approach applies to all endofunctors, whereas relation lifting is only defined for polynomials. The lifting of predicates and relations is a complex operation on first sight, but in practice it is easier to work with definitions that are based on predicate and relation lifting. For polynomial functors both approaches yield identical notions. In the following I define predicate and relation lifting and the notions of bisimulation and invariant in both, the Hermida/Jacobs and the Aczel/Mendler tradition. Then I show that for polynomial functors both definitions are equivalent.

Definition 2.6.3 (Predicate and relation lifting) Assume a polynomial functor F on the category **Set**. Predicate lifting maps F to an endofunctor on **Pred**; relation lifting maps F to an endofunctor on **Rel**. More precisely for a predicate $P \subseteq X$, and a relation $R \subseteq X \times Y$ we have

$$\mathrm{Pred}(F) : \ (P \subseteq X) \longmapsto (\mathrm{Pred}(F)(P) \ \subseteq \ F(X))$$

$$\mathrm{Rel}(F) : \ (R \subseteq X \times Y) \longmapsto (\mathrm{Rel}(F)(R) \ \subseteq \ F(X) \times F(Y))$$

Predicate and relation lifting is defined by induction on the structure of F:

- If $F(X) = X$, then

$$\begin{aligned} \mathrm{Pred}(F)(P) &= P \\ \mathrm{Rel}(F)(R) &= R \end{aligned}$$

- If $F(X) = A$ (for A a constant set), then

$$\begin{array}{rcl}
\mathrm{Pred}(F)(P) & = & \top_A \quad = \quad (A \subseteq A) \\
\mathrm{Rel}(F)(R) & = & \mathrm{Eq}(A) \quad = \quad \big(\{(a,a) \,|\, a \in A\} \subseteq A \times A\big)
\end{array}$$

- If $F(X) = F_1(X) \times F_2(X)$, then

$$\begin{array}{rcl}
\mathrm{Pred}(F)(P) & = & \mathrm{Pred}(F_1)(P) \ \times_{\mathrm{P}} \ \mathrm{Pred}(F_2)(P) \\
& = & \big\{(x,y) \mid \mathrm{Pred}(F_1)(P)(x) \ \text{ and } \ \mathrm{Pred}(F_2)(P)(y)\big\} \\
\mathrm{Rel}(F)(R) & = & \mathrm{Rel}(F_1)(R) \ \times_{\mathrm{R}} \ \mathrm{Rel}(F_2)(R) \\
& = & \big\{((x_1,x_2),(y_1,y_2)) \mid \mathrm{Rel}(F_1)(R)(x_1,y_1) \\
& & \qquad \text{and } \ \mathrm{Rel}(F_2)(R)(x_2,y_2)\big\}
\end{array}$$

- If $F(X) = F_1(X) + F_2(X)$, then

$$\begin{array}{rcl}
\mathrm{Pred}(F)(P) & = & \mathrm{Pred}(F_1)(P) \ +_{\mathrm{P}} \ \mathrm{Pred}(F_2)(P) \\
& = & \big\{\kappa_1\, x \mid \mathrm{Pred}(F_1)(P)(x)\big\} \ \cup \ \big\{\kappa_2\, y \mid \mathrm{Pred}(F_2)(P)(y)\big\} \\
\mathrm{Rel}(F)(R) & = & \mathrm{Rel}(F_1)(R) \ +_{\mathrm{R}} \ \mathrm{Rel}(F_2)(R) \\
& = & \big\{(\kappa_1\, x_1, \kappa_1\, y_1) \mid \mathrm{Rel}(F_1)(R)(x_1,y_1)\big\} \\
& & \ \cup \ \big\{(\kappa_2\, x_2, \kappa_2\, y_2) \mid \mathrm{Rel}(F_2)(R)(x_2,y_2)\big\}
\end{array}$$

- If $F(X) = A \Rightarrow F_1(X)$, then

$$\begin{array}{rcl}
\mathrm{Pred}(F)(P) & = & \top_A \ \Rightarrow_{\mathrm{P}} \ \mathrm{Pred}(F_1)(P) \\
& = & \big\{f \mid f : A \longrightarrow F_1(X) \text{ such that} \\
& & \qquad \forall a \in A \,.\, \mathrm{Pred}(F_1)(P)(f(a))\big\} \\
\mathrm{Rel}(F)(R) & = & \mathrm{Eq}(A) \ \Rightarrow_{\mathrm{R}} \ \mathrm{Rel}(F_1)(R) \\
& = & \big\{(f,g) \mid f : A \longrightarrow F_1(X), g : A \longrightarrow F_1(Y) \\
& & \qquad \text{such that } \ \forall a \in A \,.\, \mathrm{Rel}(F_1)(R)(f(a),g(a)) \ \big\}
\end{array}$$

Predicate and relation lifting works by substituting the cartesian closed structure of **Pred** or **Rel** for the the cartesian closed structure of the base category in F. For constants one substitutes truth or equality. The effect is as follows: If you think of an element $u \in F(X)$ as a structured box containing constants and elements of X, then $\mathrm{Pred}(F)(P)(u)$ holds if P holds for all the elements of X in u. Relation lifting relates two boxes $u \in F(X)$ and $v \in F(Y)$. The pair (u,v) is in the relation $\mathrm{Rel}(F)(R)$ if both u and v have the same structure and additionally constants in u and v at corresponding places are equal and elements of X and Y at corresponding places in u and v are related by R.

Example 2.6.4 For the interface functor of sequences $T_{\mathsf{Seq}}(X) = (A \times X) + \mathbf{1}$ one gets for an arbitrary predicate $P \subseteq X$ and a relation $R \subseteq X \times Y$:

$$\begin{aligned} \mathrm{Pred}(T_{\mathsf{Seq}})(P) &= \{\kappa_1(a, x) \mid a \in A \text{ and } x \in P\} \cup \{\kappa_2 *\} \\ \mathrm{Rel}(T_{\mathsf{Seq}})(R) &= \{(\kappa_1(a, x), \kappa_1(b, y)) \mid a, b \in A \wedge a = b \wedge (x, y) \in R\} \cup \{(\kappa_2 *,\ \kappa_2 *)\} \end{aligned}$$

Thus, $\mathrm{Pred}(T_{\mathsf{Seq}})(P)$ holds on $\mathsf{next}(s)$ if either s is the empty sequence or the tail of s is in P. And $(\mathsf{next}\, s,\ \mathsf{next}\, q)$ is in $\mathrm{Rel}(T_{\mathsf{Seq}})(R)$ if either *both* s and r are empty or *both* have the same first element a and, additionally, their tails are related by R. ■

Predicate and relation lifting enjoy a lot of useful properties. I consider them in the next subsection, here I continue with the definition of bisimulation and invariant.

Definition 2.6.5 (Hermida/Jacobs Invariant and Bisimulation) Let F be a polynomial functor and consider two coalgebras $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$.

- A predicate $P \subseteq X$ is called a *Hermida/Jacobs invariant* (for c) if for all $x \in X$

$$P(x) \quad \text{implies} \quad \mathrm{Pred}(F)(P)(c\, x).$$

- A relation $R \subseteq X \times Y$ is called a *Hermida/Jacobs bisimulation* (for c and d) if for all $x \in X$ and $y \in Y$,

$$R(x, y) \quad \text{implies} \quad \mathrm{Rel}(F)(R)(c\, x,\ d\, y).$$

Invariants were first called *mongruences* in (Jacobs, 1995) in analogy with congruences. (Rutten, 2000) uses the term *subsystem* for invariants.

Example 2.6.6 I described the notion of bisimulation for sequence coalgebras that comes out of the preceding definition already at the beginning of this subsection. For an example of an invariant assume that a and b are elements of A. The set of infinite sequences that always output either a or b form the invariant P_{ab}. It can be characterised as

$$s \in P_{ab} \quad \text{if and only if} \quad \big(\mathsf{next}(s) = \kappa_1(a, s') \vee \mathsf{next}(s) = \kappa_1(b, s')\big) \ \wedge s' \in P_{ab} \quad ■$$

Definition 2.6.7 (Aczel/Mendler Invariant and Bisimulation)
Assume an endofunctor T on **Set** and coalgebras $c : X \longrightarrow T(X)$ and $d : Y \longrightarrow T(Y)$.

- A predicate $P \subseteq X$ is called a *Aczel/Mendler invariant* (for c) if there exists a subcoalgebra on P, that is if there is a coalgebra $p : P \longrightarrow T(P)$ such that the inclusion $\iota : P \longrightarrow X$ is a T–coalgebra morphism $p \longrightarrow c$ (i.e., the left diagram below commutes).

- A relation $R \subseteq X \times Y$ is called an *Aczel/Mendler bisimulation* (for c and d) if there exists a coalgebra $r : R \longrightarrow T(R)$ such that the projections $\pi_1 : R \longrightarrow X$ and $\pi_2 : R \longrightarrow Y$ are T–coalgebra morphisms (i.e., if the right diagram below commutes).

$$\begin{tikzcd}
P \arrow[r, "\iota"] \arrow[d, "p"'] & X \arrow[d, "c"] \\
T(P) \arrow[r, "T(\iota)"] & T(X)
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
X \arrow[d, "c"'] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "r"'] & Y \arrow[d, "d"] \\
T(X) & T(R) \arrow[l, "T(\pi_1)"'] \arrow[r, "T(\pi_2)"] & T(Y)
\end{tikzcd}$$

There is the following standard result.

Proposition 2.6.8 *For polynomial functors the Hermida/Jacobs notions of invariant and bisimulation coincide with the Aczel/Mendler notions.*

Proof This proposition follows from the Propositions 3.4.20 and 3.4.21 (on page 102ff) on extended polynomial functors. For illustration I sketch here a proof for polynomial functors.

The fact about invariants follows from $\mathrm{Pred}(T)(P) = T(P)$, which is obvious from the definition of predicate lifting. For bisimulations consider the following diagram

$$\begin{tikzcd}
X \arrow[dd, "c"'] & R \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "r"] \arrow[ddd, bend left=80, "c\times d"] & Y \arrow[dd, "d"] \\
 & F(R) \arrow[dl, "F(\pi_1)"'] \arrow[dr, "F(\pi_2)"] \arrow[dd, "\langle F(\pi_1), F(\pi_2)\rangle" description] & \\
F(X) & & F(Y) \\
 & \mathrm{Rel}(F)(R) \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}$$

By induction on the structure of F one can show that $\langle F(\pi_1), F(\pi_2)\rangle$ restricts to an isomorphism $F(R) \longrightarrow \mathrm{Rel}(F)(R)$. So from every Hermida/Jacobs bisimulation R one can construct r as $\langle F(\pi_1), F(\pi_2)\rangle^{-1} \circ (c \times d)$ and vice versa. □

Note that the proof shows that for polynomial functors in **Set** the witness for an Aczel/Mendler invariant or bisimulation is uniquely determined. For the rest of this section I will not distinguish between the Aczel/Mendler and the Hermida/Jacobs variants of bisimulations and invariants. (The distinction will become important in Chapter 3.)

2.6.3. Properties of Bisimulations and Invariants

All the following results about bisimulations and invariants have been taken from the literature, especially from (Rutten, 2000). However, I prefer to give different proofs that match the proofs in Section 3.4. The first lemma collects properties of predicate and relation lifting.

Lemma 2.6.9 *Let F be a polynomial functor on* **Set**.

1. *Predicate lifting and relation lifting is monotone, that is for predicates $P, Q \subseteq X$ and relations $R, S \subseteq X \times T$ we have*

$$\begin{array}{rcl} P \subseteq Q & \textit{implies} & \mathrm{Pred}(F)(P) \subseteq \mathrm{Pred}(F)(Q) \\ R \subseteq S & \textit{implies} & \mathrm{Rel}(F)(R) \subseteq \mathrm{Rel}(F)(R) \end{array}$$

   So Pred(F) *and* Rel(F) *are endofunctors on* **Pred** *and* **Rel**, *respectively.*

2. *Predicate lifting commutes with truth; relation lifting commutes with equality and $(-)^{\mathrm{op}}$:*

$$\begin{array}{rcl} \mathrm{Pred}(F)(\top_X) & = & \top_{F(X)} \\ \mathrm{Rel}(F)(\mathrm{Eq}(X)) & = & \mathrm{Eq}(F(X)) \\ \mathrm{Rel}(F)(R^{\mathrm{op}}) & = & (\mathrm{Rel}(F)(R))^{\mathrm{op}} \end{array}$$

3. *If R is symmetric then also* Rel(F)(R) *is symmetric.*

4. *Predicate and relation lifting preserve arbitrary conjunctions. For a collection of predicates$(P_i)_{i \in i}$ and a collection of relation $(R_i)_{i \in I}$ we have*

$$\begin{array}{rcl} \bigwedge_i \mathrm{Rel}(F)(R_i) & = & \mathrm{Rel}(F)\big(\bigwedge_i R_i\big) \\ \bigwedge_i \mathrm{Pred}(F)(P_i) & = & \mathrm{Pred}(F)\big(\bigwedge_i P_i\big) \end{array}$$

5. *For disjunction we have*

$$\begin{array}{rcl} \bigvee_i \mathrm{Rel}(F)(R_i) & \subseteq & \mathrm{Rel}(F)\big(\bigvee_i R_i\big) \\ \bigvee_i \mathrm{Pred}(F)(P_i) & \subseteq & \mathrm{Pred}(F)\big(\bigvee_i P_i\big) \end{array}$$

6. *Relation lifting commutes with relational composition in the following sense*

$$\mathrm{Rel}(F)(R \circ S) \quad = \quad \mathrm{Rel}(F)(R) \circ \mathrm{Rel}(F)(S)$$

   for relations $R \subseteq X \times Y$ and $S \subseteq Y \times Z$.

7. *Predicate and relation lifting (of F) are fibred (over F). That is, assuming functions $f : U \longrightarrow X$, $g : V \longrightarrow Y$, a predicate $P \subseteq X$, and a relation $R \subseteq X \times Y$:*

$$\begin{array}{rcl} \mathrm{Pred}(F)(f^* P) & = & (F(f))^* \, \mathrm{Pred}(F)(P) \\ \mathrm{Rel}(F)((f \times g)^* R) & = & (F(f) \times F(g))^* \, \mathrm{Rel}(F)(R) \end{array}$$

Proof Item 3 follows from Item 2. The other items are proved by induction on the structure of F. In the induction steps one uses the lemmas of Section 2.4.3. For Item 1 Lemma 2.4.5, for Item 2 the Lemmas 2.4.7, 2.4.8, and 2.4.11, for Item 4 Lemma 2.4.9, for Item 5 the Lemmas 2.4.10 and 2.4.5, for Item 6 Lemma 2.4.12, and finally for Item 7 Lemma 2.4.15. □

These results about relation lifting give rise to the following results about bisimulation and invariants.

Proposition 2.6.10 *Consider three coalgebras $c : X \longrightarrow F(X)$, $d : Y \longrightarrow F(Y)$, and $e : Z \longrightarrow F(Z)$ for a polynomial functor F.*

1. $\mathrm{Eq}(X)$ *is a bisimulation and $\top_X$ an invariant.*
2. *Invariants are closed under union and intersection: for an arbitrary collection of invariants $(P_i)_{i \in I}$ both $\bigwedge_{i \in I} P_i$ and $\bigvee_{i \in I} P_i$ are invariants.*
3. *Similarly for bisimulations: for a collection of bisimulations $(R_i)_{i \in I}$ both $\bigwedge_{i \in I} R_i$ and $\bigvee_{i \in I} R_i$ are bisimulations.*
4. *The relation R is a bisimulation for c and d if and only if R^{op} is a bisimulation for d and c.*
5. *If R is a bisimulation for c and d and S is a bisimulation for d and e then $R \circ S$ is a bisimulation for c and e.*

Proof Apply Lemma 2.6.9. Consider for instance the union of bisimulations (Item 3). From $(x, y) \in \bigvee_i R_i$ and the assumptions we get that there is some $j \in I$ such that $(c\,x,\, d\,y) \in \mathsf{Rel}(F)(R_j)$. Hence $(c\,x,\, d\,y) \in \bigvee_i \mathrm{Rel}(F)(R_i)$ and Lemma 2.6.9 (5) implies $(c\,x,\, d\,y) \in \mathrm{Rel}(F)(\bigvee_i R_i)$, so $\bigvee_i R_i$ is a bisimulation for c and d. □

The Items 2 and 3 of the preceding Proposition imply that both invariants and bisimulations for coalgebras of polynomial functors form a complete lattice. In particular for any two coalgebras $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$ on the same interface functor there exist a greatest bisimulation for c and d. This greatest bisimulation is called *bisimilarity* and denoted with ${}_c\!\underline{\leftrightarrow}_d$.

Proposition 2.6.11 *Bisimilarity ${}_c\!\underline{\leftrightarrow}_c$ on one coalgebra is an equivalence relation.*

Proof One proves ${}_c\!\underline{\leftrightarrow}_d \circ {}_d\!\underline{\leftrightarrow}_e \subseteq {}_c\!\underline{\leftrightarrow}_e$ and $({}_d\!\underline{\leftrightarrow}_c)^{\mathrm{op}} \subseteq {}_c\!\underline{\leftrightarrow}_d$ with Proposition 2.6.10. □

2.6.4. Bisimulations and Coalgebra Morphisms

This subsection relates bisimulations and coalgebra morphisms.

Proposition 2.6.12 *Let $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$ be two coalgebras for a polynomial functor F. A function $f : X \longrightarrow Y$ is a morphism between c and d if and only if the graph of f given by $\mathsf{graph}(f) = \coprod_{\mathrm{id}_X \times f} \mathrm{Eq}(X)$ is a bisimulation for c and d.*

Proof I have to show that for a function $f : X \longrightarrow Y$ an arbitrary pair $(c\,x, d(f\,x))$ is in $\mathrm{Rel}(F)(\mathsf{graph}(f))$ if and only if $F(f)(c\,x) = d(f\,x)$. The latter is equivalent with $(c\,x, d(f\,x)) \in (F(f) \times \mathrm{id}_{F(Y)})^* \,\mathrm{Eq}(F(Y))$.

First note that in **Set** there is an equivalence

$$\coprod_{\mathrm{id}_X \times f} \mathrm{Eq}(X) \quad = \quad (f \times \mathrm{id}_Y)^* \,\mathrm{Eq}(Y) \tag{2.7}$$

Using this intermediate result I compute

$$\begin{array}{lll} & \mathrm{Rel}(F)(\mathsf{graph}(f)) & \\ = & \mathrm{Rel}(F)\big((f \times \mathrm{id}_Y)^* \,\mathrm{Eq}(Y)\big) & \text{by (2.7)} \\ = & (F(f) \times F(\mathrm{id}_Y))^* \,\mathrm{Rel}(F)(\mathrm{Eq}(Y)) & \text{by 2.6.9 (7)} \\ = & (F(f) \times \mathrm{id}_{F(Y)})^* \,\mathrm{Eq}(F(Y)) & \text{by 2.6.9 (2)} \quad \square \end{array}$$

The preceding proposition justifies the informal claim that coalgebra morphisms preserve the observable behaviour. It implies that, for a coalgebra morphism f, it holds that $x \;{}_c\!\underline{\leftrightarrow}_d\; f\,x$.

The kernel of a function $f : X \longrightarrow Y$ is a binary relation on X and contains precisely those pairs (x_1, x_2) for which $f(x_1) = f(x_2)$. The kernel can be characterised as $\mathsf{ker}(f) = \mathsf{graph}(f) \circ \mathsf{graph}(f)^{\mathrm{op}}$. The next result is an immediate consequence of the Propositions 2.6.12 and 2.6.10 (5).

Proposition 2.6.13 *For every polynomial functor F the kernel of an arbitrary morphism between F–coalgebras is a bisimulation.* □

2.6.5. Relating Bisimulations and Invariants

It is often useful to combine bisimulations and invariants to obtain new bisimulations and/or invariants. The following two propositions are taken from (Jacobs, 1997b). Let $R \subseteq X \times Y$ be a relation. Then $\coprod_{\pi_1} R = \{x \in X \mid \exists y \in Y \,.\, R(x,y)\}$ is a predicate on X. Proposition 2.6.15 states that if R is a bisimulation, then $\coprod_{\pi_1} R$ is an invariant.

A second construction is $R \wedge \pi_1^* \,P = \{(x,y) \mid R(x,y) \wedge P(x)\}$. Proposition 2.6.17 shows that for a bisimulation R and an invariant P the relation $R \wedge \pi_1^* \,P$ is a bisimulation again. For both propositions adequate properties of relation lifting are necessary. I need these lemmas about predicate and relation lifting in Section 3.4.4 (starting on page 99) again when I prove the validity of the two mentioned constructions for extended polynomial functors.

Lemma 2.6.14 *Let $R \subseteq X \times Y$ be an arbitrary relation and F be a polynomial functor. Then*

$$\textstyle\coprod_{\pi_1} \mathrm{Rel}(F)(R) \quad = \quad \mathrm{Pred}(F)(\coprod_{\pi_1} R)$$

Proof The proof goes by induction on the structure of F. In the induction steps one uses Lemma 2.4.13. For instance for $F = F_1 \times F_2$:

$$\begin{aligned}
&\textstyle\coprod_{\pi_1} \mathrm{Rel}(F_1 \times F_2)(R) \quad = && \\
&\quad = \quad \textstyle\coprod_{\pi_1} \big(\mathrm{Rel}(F_1)(R) \; \times_{\mathrm{R}} \; \mathrm{Rel}(F_1)(R)\big) && \\
&\quad = \quad \big(\textstyle\coprod_{\pi_1} \mathrm{Rel}(F_1)(R)\big) \; \times_{\mathrm{P}} \; \big(\textstyle\coprod_{\pi_1} \mathrm{Rel}(F_2)(R)\big) && \text{by 2.4.13} \\
&\quad = \quad \mathrm{Pred}(F_1)(\textstyle\coprod_{\pi_1} R) \; \times_{\mathrm{P}} \; \mathrm{Pred}(F_2)(\textstyle\coprod_{\pi_1} R) && \text{by Ind. Hyp.} \\
&\quad = \quad \mathrm{Pred}(F_1 \times F_2)(\textstyle\coprod_{\pi_1} R) && \square
\end{aligned}$$

Proposition 2.6.15 *Consider two F–coalgebras $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$, and a relation $R \subseteq X \times Y$. If R is a bisimulation for c and d then $\coprod_{\pi_1} R$ is an invariant for c.*

Proof Assume that R is a bisimulation for c and d. Define $P \stackrel{\mathrm{def}}{=} \coprod_{\pi_1} R$. I have to show that for all $x \in P$ also $c(x) \in \mathrm{Pred}(F)(P)$ holds. Assuming $x \in P$ there exists $y \in Y$ such that $R(x, y)$ and because R is a bisimulation I have $(c(x), d(y)) \in \mathrm{Rel}(F)(R)$. With Lemma 2.6.14 $c(x) \in \mathrm{Pred}(F)(\coprod_{\pi_1} R)$ follows. □

Now towards the second construction.

Lemma 2.6.16 *Let F be a polynomial functor, S and R arbitrary relations, and P and Q arbitrary predicates. Then*

$$\mathrm{Rel}(F)(R) \; \wedge \; \pi_1^* \big(\mathrm{Pred}(F)(P)\big) \quad = \quad \mathrm{Rel}(F)(R \wedge \pi_1^* P)$$

Proof The proof proceeds by induction on the structure of F. The induction steps exploit Lemma 2.4.14. For the case of the product we have for instance

$$\begin{aligned}
&\mathrm{Rel}(F_1 \times F_2)(R) \; \wedge \; \pi_1^* \big(\mathrm{Pred}(F_1 \times F_2)(P)\big) && \\
&\quad = \quad \big(\mathrm{Rel}(F_1)(R) \; \times_{\mathrm{R}} \; \mathrm{Rel}(F_2)(R)\big) \quad \wedge && \\
&\qquad\qquad \pi_1^* \big(\mathrm{Pred}(F_1)(P) \; \times_{\mathrm{P}} \; \mathrm{Pred}(F_2)(P)\big) && \\
&\quad = \quad \big(\mathrm{Rel}(F_1)(R) \; \times_{\mathrm{R}} \; \mathrm{Rel}(F_2)(R)\big) \quad \wedge && \text{by 2.4.14} \\
&\qquad\qquad \big(\pi_1^* \,(\mathrm{Pred}(F_1)(P)) \; \times_{\mathrm{R}} \; \pi_1^* \,(\mathrm{Pred}(F_2)(P))\big) && \\
&\quad = \quad \big(\mathrm{Rel}(F_1)(R) \; \wedge \; \pi_1^* \big(\mathrm{Pred}(F_1)(P)\big)\big) \; \times_{\mathrm{R}} && \text{by 2.4.9 (2)} \\
&\qquad\qquad \big(\mathrm{Rel}(F_2)(R) \; \wedge \; \pi_1^* \big(\mathrm{Pred}(F_2)(P)\big)\big) && \\
&\quad = \quad \mathrm{Rel}(F_1)(R \wedge \pi_1^* P) \; \times_{\mathrm{R}} \; \mathrm{Rel}(F_2)(R \wedge \pi_1^* P) && \text{by Ind. Hyp.} \\
&\quad = \quad \mathrm{Rel}(F_1 \times F_2)(R \wedge \pi_1^* P) && \square
\end{aligned}$$

Proposition 2.6.17 *Consider two F–coalgebras $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$, a predicate $P \subseteq X$, and a relation $R \subseteq X \times Y$. If P is an invariant for c and R is a bisimulation for c and d then also $R \wedge \pi_1^* P = \{(x, y) \mid R(x, y) \wedge P(x)\}$ is a bisimulation for c and d.*

Proof Assume $x \in X$ with $P(x)$ and $y \in Y$ with $R(x, y)$. From the assumptions that P is an invariant and R is a bisimulation I know that $c(x) \in \mathrm{Pred}(F)(P)$ and that $(c(x), d(y)) \in \mathrm{Rel}(F)(R)$. With Lemma 2.6.16 I conclude that indeed $(c(x), d(y)) \in \mathrm{Rel}(F)(R \wedge \pi_1^* P)$. □

It is now possible to derive more results on a higher level of reasoning.

Proposition 2.6.18 *Let $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$ be two coalgebras for a polynomial functor F and let $f : c \longrightarrow d$ be a coalgebra morphism.*

1. *If $P \subseteq X$ is an invariant for c then $\coprod_f P$ is an invariant for d.*
2. *If $Q \subseteq Y$ is an invariant for d then $f^* Q$ is an invariant for c.*

Proof The graph of f is given as $\coprod_{\mathrm{id}_X \times f} \mathrm{Eq}(X) = (f \times \mathrm{id}_Y)^* \mathrm{Eq}(Y)$ as a relation over $X \times Y$. Proposition 2.6.12 shows that this relation is a bisimulation. If P is an invariant then, because of Proposition 2.6.17, $\coprod_{\mathrm{id}_X \times f} \mathrm{Eq}(X) \wedge \pi_1^* P$ is a bisimulation too. Proposition 2.6.15 shows then that $\coprod_{\pi_2} \big(\coprod_{\mathrm{id}_X \times f} \mathrm{Eq}(X) \wedge \pi_1^* P\big)$ is an invariant. Now I compute

$$\begin{array}{lll}
\multicolumn{3}{l}{\coprod_{\pi_2} \big(\coprod_{\mathrm{id}_X \times f} \mathrm{Eq}(X) \wedge \pi_1^* P\big)} \\
= & \coprod_{\pi_2} \coprod_{\mathrm{id}_X \times f} \big(\mathrm{Eq}(X) \wedge (\mathrm{id}_X \times f)^* \pi_1^* P\big) & \text{by Frobenius} \\
= & \coprod_f \coprod_{\pi_2} \big(\coprod_\delta \top_X \wedge \pi_1^* P\big) & \\
= & \coprod_f \coprod_{\pi_2} \coprod_\delta \big(\top_X \wedge \delta^* \pi_1^* P\big) & \text{by Frobenius} \\
= & \coprod_f (\top_X \wedge P) & \\
= & \coprod_f P &
\end{array}$$

This proves Item 1. For Item 2 I obtain from the same lemmas that

$$\coprod_{\pi_1} \big((f \times \mathrm{id}_Y)^* \mathrm{Eq}(Y) \wedge \pi_2^* Q\big)$$

is an invariant. Then

$$\begin{array}{lll}
\multicolumn{3}{l}{\coprod_{\pi_1} \big((f \times \mathrm{id}_Y)^* \mathrm{Eq}(Y) \wedge \pi_2^* Q\big)} \\
= & \coprod_{\pi_1} \big((f \times \mathrm{id}_Y)^* \coprod_\delta \top_Y \wedge (f \times \mathrm{id}_Y)^* \pi_2^* Q\big) & \\
= & \coprod_{\pi_1} (f \times \mathrm{id}_Y)^* \big(\coprod_\delta \top_Y \wedge \pi_2^* Q\big) & \text{fibred } \wedge \\
= & \coprod_{\pi_1} (f \times \mathrm{id}_Y)^* \coprod_\delta \big(\top_Y \wedge \delta^* \pi_2^* Q\big) & \text{by Frobenius} \\
= & \coprod_{\pi_1} \coprod_{\langle \mathrm{id}_X, f\rangle} f^* \big(\top_Y \wedge \delta^* \pi_2^* Q\big) & \text{by BC} \\
= & f^* Q & \square
\end{array}$$

The next result appears as Proposition 6.2 in (Rutten, 2000).

Proposition 2.6.19 *Let $c : X \longrightarrow F(X)$ be a coalgebra of a polynomial functor F. A predicate $P \subseteq X$ is an invariant for c if and only if the diagonal on P, the relation $\coprod_\delta P$, is a bisimulation for c.*

Proof If $\coprod_\delta P$ is a bisimulation, then $\coprod_{\pi_1} \coprod_\delta P = P$ is an invariant by Proposition 2.6.15. If, for the other direction, P is an invariant, then by Frobenius $\coprod_\delta P = \mathrm{Eq}(X) \wedge \pi^* P$ and $\coprod_\delta P$ is a bisimulation by Propositions 2.6.17 and 2.6.10 (1). □

2.6.6. Least and Greatest Invariants

Let $\alpha : X \longrightarrow F(X)$ be an arbitrary coalgebra for a polynomial functor F. Proposition 2.6.10 (2) shows that the invariants for α form a complete lattice. In particular there exists, for any predicate $P \subseteq X$, the greatest invariant contained in P, to be denoted with $\underline{P}$, and the least invariant containing P, denoted with $\overline{P}$. Obviously

$$\begin{array}{rcl} \overline{P} & = & \bigwedge \{Q \mid P \subseteq Q \text{ and } Q \text{ is an invariant }\} \\ \underline{P} & = & \bigvee \{Q \mid Q \subseteq P \text{ and } Q \text{ is an invariant }\} \end{array}$$

and also

$$P \subseteq Q \qquad \text{implies} \qquad \overline{P} \subseteq \overline{Q} \qquad \text{and} \qquad \underline{P} \subseteq \underline{Q}$$

It is possible to get alternative characterisations for the greatest invariant $\underline{P}$ via the Knaster/Tarski fixed point theorem (Tarski, 1955). Consider the following endofunctor on the fibre $\mathbf{Pred}_X$ for a fixed predicate $P \subseteq X$:

$$\Phi_P(Q \subseteq X) \stackrel{\text{def}}{=} P \wedge \alpha^* \,\mathrm{Pred}(F)(Q)$$

Any fixed point of Φ_P is an invariant implying P. For an invariant Q smaller than P it holds that $Q \subseteq \Phi_P(Q)$. Therefore the Knaster/Tarski theorem implies

$$\begin{array}{rcl} \underline{P} & = & \bigvee \{Q \mid Q \subseteq \Phi_P(Q)\} \\ & = & \bigvee \{Q \mid Q \subseteq P \wedge \alpha^* \,\mathrm{Pred}(F)(Q)\} \end{array} \qquad (*)$$

Consider now the descending chain $(P_i)_{i \in \mathbb{N}}$ defined as

$$P_0 \stackrel{\text{def}}{=} \top_X \qquad\qquad P_{i+1} \stackrel{\text{def}}{=} \Phi_P(P_i)$$

So $P_1 = P$ and $P_2 = P \wedge \alpha^* \,\mathrm{Pred}(F)(P)$ and so on. The limit $\bigwedge_i P_i$ of the chain is a fixed point for Φ_P because

$$\begin{array}{rcll} \Phi_P(\bigwedge_i P_i) & = & P \wedge \alpha^* \,\mathrm{Pred}(F)(\bigwedge_i P_i) & \\ & = & P \wedge \alpha^* \bigwedge_i \mathrm{Pred}(F)(P_i) & \text{by 2.6.9 (4)} \end{array}$$

$$\begin{aligned} &= P \wedge \textstyle\bigwedge_i \big(\alpha^* \operatorname{Pred}(F)(P_i)\big) && (\dagger) \\ &= \textstyle\bigwedge_i \big(P \wedge \alpha^* \operatorname{Pred}(F)(P_i)\big) \\ &= P_0 \wedge \textstyle\bigwedge_i P_{i+1} \quad = \quad \textstyle\bigwedge_i P_i \end{aligned}$$

The line (†) follows from the fact that I–indexed conjunctions are fibred (see page 37). So $\bigwedge_i P_i$ is a fixed point of Φ_P. To show that it is the largest one, assume a fixed point $Q = \Phi_P(Q)$. Because $Q \subseteq \top_X$ we have also $Q = \Phi_P(Q) \subseteq \Phi_P(\top_X) = P_1$ and by induction $Q \subseteq P_i$ for all i. Therefore $Q \subseteq \bigwedge_i P_i$ and finally

$$\underline{P} \quad = \quad \textstyle\bigwedge_i P_i$$

This last characterisation for the greatest invariant $\underline{P}$ corresponds to the intuitive way of computing $\underline{P}$ from P: Starting from P one deletes all those $x \in P$ that have a successor state $y \notin P$. In (Jacobs, 1997b) the greatest invariant is computed as the limit of the chain

$$P'_0 \quad \stackrel{\text{def}}{=} \quad P \qquad\qquad P'_{i+1} \quad \stackrel{\text{def}}{=} \quad P'_i \wedge \alpha^* \operatorname{Pred}(F)(P'_n)$$

This is essentially the same, because $P'_i = P_{i+1}$.

One can also present least invariants as fixed points of an endofunctor on $\mathbf{Pred}_X$ and also as the limit of an ascending chain of predicates. Such a presentation justifies the intuitive way of obtaining $\overline{P}$ from P: Starting from $x \in P$ one adds all reachable successor states of x to P. However to construct the ascending chain in a general way one needs the left adjoint of Pred(F) (considered as a functor $\mathbf{Pred}_X \longrightarrow \mathbf{Pred}_{F(X)}$). This has been done in (Jacobs, 1997b), I would like to omit the technicalities here.

2.6.7. Final Coalgebras

A T–coalgebra γ is final if it is the final object in the category of T–coalgebras. This is the case if for any coalgebra α there exists exactly one coalgebra morphism $!_\alpha : \alpha \longrightarrow \gamma$. The final coalgebra is, if it does exist, an isomorphism. Further, any two final coalgebras are isomorphic (as objects in $CoAlg(T)$).

I explained before that an element x of the state space of a T–coalgebra can be considered as an automaton. Assume that $\gamma : Z \longrightarrow T(Z)$ is the final T–coalgebra. The unique morphism ! into the final coalgebra maps x to an automaton in Z. Coalgebra morphisms preserve the observable behaviour. So any possible state of any T–automaton can be mapped to a state in Z with the same behaviour. Because of the uniqueness aspect of finality the automaton γ is minimal: There are no two different states in Z that show the same behaviour. These properties suggest to consider the state space of the final T–coalgebra as the type of all possible behaviours of T–coalgebras. A type that is defined in this way as the state space of some final coalgebra is called a *behavioural type* in this thesis.

For final coalgebras there are the following results in (Rutten, 2000).

Theorem 2.6.20 *For all polynomial functors F on* **Set** *there exists a final F–coalgebra.*

There are more general results available, see (Kawahara and Mori, 2000; Aczel and Mendler, 1989).

Proof (Sketch) Polynomial functors are continuous, therefore the final F–coalgebra can be obtained as limit of the chain

$$\begin{tikzcd} \mathbf{1} & F(\mathbf{1}) \arrow[l, "!"'] & F^2(\mathbf{1}) \arrow[l, "F(!)"'] & F^3(\mathbf{1}) \arrow[l, "F^2(!)"'] & F^4(\mathbf{1}) \arrow[l, "F^3(!)"'] & \cdots \end{tikzcd} \qquad \Box$$

Proposition 2.6.21 *Let F be a polynomial functor. The bisimilarity on the final F–coalgebra $\gamma : Z \longrightarrow F(Z)$ is contained in the equality: ${}_{\gamma}\underline{\leftrightarrow}_{\gamma} \subseteq \mathrm{Eq}(Z)$.*

Together with Proposition 2.6.10 (1) this means that for two states $x, y \in Z$ we have $x \;{}_{\gamma}\underline{\leftrightarrow}_{\gamma}\; y$ if and only if $x = y$.

Proof Consider ${}_{\gamma}\underline{\leftrightarrow}_{\gamma}$ as a set of pairs over Z. Proposition 2.6.8 shows that there is a coalgebra $r : {}_{\gamma}\underline{\leftrightarrow}_{\gamma} \longrightarrow F({}_{\gamma}\underline{\leftrightarrow}_{\gamma})$ such that the projections $Z \xleftarrow{\pi_1} {}_{\gamma}\underline{\leftrightarrow}_{\gamma} \xrightarrow{\pi_2} Z$ are coalgebra morphisms $r \longrightarrow \gamma$. Finality of γ yields $\pi_1 = \pi_2$ so $x = \pi_1(x, y) = \pi_2(x, y) = y$ for any pair $x \;{}_{\gamma}\underline{\leftrightarrow}_{\gamma}\; y$. □

With the propositions 2.6.12 and 2.6.10 (5) we obtain the following Corollary.

Corollary 2.6.22 *Consider two coalgebras $c : X \longrightarrow F(X)$ and $d : Y \longrightarrow F(Y)$ for a polynomial functor F. Two states $x \in X$ and $y \in Y$ are bisimilar (i.e., $x \;{}_{c}\underline{\leftrightarrow}_{d}\; y$) if and only if x and y have equal images in the final F–coalgebra (i.e., $!_c(x) = !_d(y)$).* □

In the remainder of this section I show how finality can be exploited in the example of sequences. Let me first construct the final sequence coalgebra to get an impression about how final coalgebras look like. The state space $\mathsf{Seq}[A]$ of the final sequence coalgebra is given by

$$\mathsf{Seq}[A] \quad = \quad \{f : \mathbb{N} \longrightarrow A + \mathbf{1} \mid \forall n \in \mathbb{N} \,.\, f(n) = \bot \;\text{ implies }\; \forall m > n \,.\, f(m) = \bot\}$$

where $\bot = \kappa_2 *$. The final sequence coalgebra $\mathsf{next} : \mathsf{Seq}[A] \longrightarrow T_{\mathsf{Seq}}(\mathsf{Seq}[A])$ is given by

$$\mathsf{next}(f) \quad = \quad \begin{cases} \bot & \text{if } f(0) = \bot \\ \kappa_1(a,\ \lambda n \,.\, f(n+1)) & \text{if } f(0) = \kappa_1 a \end{cases} \tag{2.8}$$

To show that next is indeed the final coalgebra I have to construct a coalgebra morphism $!_\alpha : \alpha \longrightarrow \mathsf{next}$ for every sequence coalgebra $\alpha : X \longrightarrow T_{\mathsf{Seq}}(X)$. For that I need an utility function $\alpha^n : X \longrightarrow A + 1$:

$$\alpha^0(x) \quad = \quad \begin{cases} \bot & \text{if } \alpha\, x = \bot \\ \kappa_1\, a & \text{if } \alpha\, x = \kappa_1(a, x') \end{cases}$$

$$\alpha^{n+1}(x) \quad = \quad \begin{cases} \bot & \text{if } \alpha(x) = \bot \\ \alpha^n(x') & \text{if } \alpha(x) = \kappa_1(a, x') \end{cases}$$

Now

$$!_\alpha(x) \quad = \quad \lambda n \in \mathbb{N} \,.\, \alpha^n(x) \tag{2.9}$$

From the definition of $!_\alpha$ it is clear that $!_\alpha\, x \in \mathsf{Seq}[A]$. It remains to show that $!_\alpha$ lets Diagram 2.6 commute. Assume an arbitrary $x \in X$, then

$$\begin{aligned} \mathsf{next}(!_\alpha\, x) \quad &= \quad \begin{cases} \bot & \text{if } \alpha^0\, x = \bot \\ \kappa_1(a,\ \lambda n \,.\, \alpha^{n+1}\, x) & \text{if } \alpha^0\, x = \kappa_1 a \end{cases} \\ &= \quad \begin{cases} \bot & \text{if } \alpha\, x = \bot \\ \kappa_1(a,\ !_\alpha(x')) & \text{if } \alpha\, x = \kappa_1(a, x') \end{cases} \end{aligned}$$

So $!_\alpha$ is indeed a coalgebra morphism. For the uniqueness of $!_\alpha$ assume a second coalgebra morphism $g : \alpha \longrightarrow \mathsf{next}$. Then for any $x \in X$ and $n \in \mathbb{N}$

$$\begin{aligned} g\, x\, n \quad &= \quad \mathsf{next}^n(g\, x) && \text{by definition of } \mathsf{next}^n \\ &= \quad \alpha^n\, x && g \text{ is a coalgebra morphism} \\ &= \quad !_\alpha\, x\, n \end{aligned}$$

where next^n is defined in the same way as α^n. Thus $g = !_\alpha$.

The preceding example of sequences is very typical. As above, the state space of the final coalgebra is often a set of functions. The difficulties in proving finality lie in the construction of the final coalgebra and in the construction of the mediating morphism $!_\alpha$. Proving uniqueness is usually easy.

The final coalgebra admits *coinduction*, both as a definition and as a proof principle. As a definition principle coinduction allows one to construct functions *into* the final coalgebra, that is, to construct states of the final coalgebra. The coinduction proof principle allows one to proof that two state of the final coalgebra are equal. Let me explain these two principles on the example of sequences.

There is precisely one empty sequence in $\mathsf{Seq}[A]$. To construct it consider the sequence coalgebra

$$\mathsf{bot} \ : \ \mathbf{1} \longrightarrow T_{\mathsf{Seq}}(\mathbf{1}) \ : \ * \longmapsto \bot$$

The unique sequence coalgebra morphism $!_{\mathsf{bot}}$ maps $*$ to an element in $\mathsf{Seq}[A]$ — the empty sequence denoted with empty.

As a second example for the definition principle I define the interleaving of two sequences. Note that this cannot be done by induction (or by a recursive function) because both sequences might be infinitely long. Consider the following T_{Seq}–coalgebra on $\mathsf{Seq}[A] \times \mathsf{Seq}[A]$.

$$\mathsf{next}_{\mathrm{mix}}(s, q) \quad = \quad \begin{cases} \bot & \text{if } \ \mathsf{next}(s) = \mathsf{next}(q) = \bot \\ \kappa_1(a, (s, q')) & \text{if } \ \mathsf{next}(s) = \bot \ \wedge \ \mathsf{next}(q) = \kappa_1(a, q') \\ \kappa_1(a, (q, s')) & \text{if } \ \mathsf{next}(s) = \kappa_1(a, s') \end{cases}$$

The finality of $\mathsf{Seq}[A]$ provides a coalgebra morphism $\mathsf{mix} : \mathsf{Seq}[A] \times \mathsf{Seq}[A] \longrightarrow \mathsf{Seq}[A]$ and the commutation of Diagram 2.6 corresponds to the equation

$$\mathsf{next}(\mathsf{mix}(s,q)) \quad = \quad \begin{cases} \bot & \text{if } \ \mathsf{next}(s) = \mathsf{next}(q) = \bot \\ \mathsf{next}(\mathsf{mix}(q,s)) & \text{if } \ \mathsf{next}(s) = \bot \wedge \mathsf{next}(q) \neq \bot \\ \kappa_1(a, \mathsf{mix}(q,s')) & \text{if } \ \mathsf{next}(s) = \kappa_1(a,s') \end{cases}$$

If the definition of mix meets our intuition, it should be possible to proof, for an arbitrary sequence $s \in \mathsf{Seq}[A]$, that

$$\mathsf{mix}(\mathsf{empty}, s) \quad = \quad s$$

The coinduction proof principle that allows one to prove this equation is as follows: One constructs a bisimulation that relates both s and $\mathsf{mix}(\mathsf{empty}, s)$, then by Proposition 2.6.21 it follows that they are equal. A suitable bisimulation for this equation is for instance

$$s \, R \, q \quad \text{if and only if} \quad s = \mathsf{mix}(\mathsf{empty}, q)$$

for all $s, q \in \mathsf{Seq}[A]$.

It is possible to formulate the coinduction proof principle for an arbitrary fibration. The general formulation is in (Hermida and Jacobs, 1998). Let me show how this looks like for the fibration $\begin{smallmatrix}\mathbf{SRel}\\ \downarrow \\ \mathbf{Set}\end{smallmatrix}$ of single carrier binary relations. The relation lifting of T_{Seq} gives the functor $\mathrm{Rel}(T_{\mathsf{Seq}})$ on $\mathbf{SRel}$, defined as

$$\begin{aligned} \mathrm{Rel}(T_{\mathsf{Seq}}) \quad &: \quad \big(R \subseteq X \times X\big) \longrightarrow \big(\mathrm{Rel}(T_{\mathsf{Seq}})(R) \ \subseteq \ T_{\mathsf{Seq}}(X) \times T_{\mathsf{Seq}}(X)\big) \\ \mathrm{Rel}(T_{\mathsf{Seq}})(R) \quad &= \quad (\mathrm{Eq}(A) \times_{\mathrm{R}} R) \ +_{\mathrm{R}} \ \mathrm{Eq}(\mathbf{1}) \end{aligned}$$

A coalgebra for $\mathrm{Rel}(T_{\mathsf{Seq}})$ on a relation R consists of a T_{Seq}–coalgebra α on X together with the following implication

$$s \, R \, q \qquad \text{implies} \qquad \begin{cases} \alpha(s) = \alpha(q) = \bot \quad \text{or} \\ \exists a \in A \,.\, \alpha(s) = \kappa_1(a,s') \ \wedge \ \alpha(q) = \kappa_1(a,q') \ \wedge \ s' \, R \, q' \end{cases} \qquad (*)$$

So α forms a $\mathrm{Rel}(T_{\mathsf{Seq}})$–coalgebra on R in $\mathbf{SRel}$ if and only if R is a bisimulation for α.

Hermida and Jacobs prove in (Hermida and Jacobs, 1998) the existence of a final object preserving functor $\mathit{CoAlg}(T_{\mathsf{Seq}}) \longrightarrow \mathit{CoAlg}(\mathrm{Rel}(T_{\mathsf{Seq}}))$. This functor maps the sequence coalgebra next to a coalgebra on the carrier $\mathrm{Eq}(\mathsf{Seq}[A])$. Assume we have a relation $R \subseteq \mathsf{Seq}[A] \times \mathsf{Seq}[A]$. If we can prove $(*)$ for R, then there is a $\mathrm{Rel}(T_{\mathsf{Seq}})$ coalgebra with state space R in $\mathbf{SRel}$. By the finality of the coalgebra on $\mathrm{Eq}(\mathsf{Seq}[A])$ we get a (vertical) morphism $R \longrightarrow \mathrm{Eq}(\mathsf{Seq}[A])$. This latter morphism corresponds to a proof of the following statement: $\forall x, y \,.\, x \, R \, y$ implies $x = y$. In other words the coinduction principle holds in $\mathbf{SRel}$: to prove that two sequences are equal, it is enough to construct a bisimulation that relates both.

2.7. Summary

This chapter introduced the standard concepts of categories, fibrations, algebras and coalgebras. Further it investigated the fibration of predicates **Pred** and the fibration of relations **Rel** in detail.

The results listed in Section 2.4 serve as a tool box for the next chapter. All the results about predicates and relations can be roughly divided into three groups: First those results that hold without restriction for all of product, coproduct, and exponent. This includes the lemmas about truth (Lemma 2.4.7), equality (2.4.8), and opposite relation (2.4.11). The second group contains those properties that hold for exponents only in a restricted version. Examples are the lemmas about conjunction (Lemma 2.4.9), composition (2.4.12), and (co–)fibredness (2.4.17). The third group contains Lemma 2.4.10 about disjunction.

The separation into three groups leads to different levels of generalisation of polynomial functors in the next chapter. Group one leads to properties that hold for the most general form of *higher-order polynomial functors* (discussed in Section 3.2 on page 79ff). Properties that depend on the second group hold only for the intermediate generalisation of *extended polynomial functors* (discussed in Section 3.4 on page 93ff).

The two results about the union of bisimulations and invariants depend on the third group (i.e., on Lemma 2.4.10). Therefore straightforward generalisations of these two results do not hold for any generalisation of polynomial functors considered in the present thesis. To obtain the result about the union of invariants one has to choose a different generalisation of the notion of invariant (see Subsection 3.4.6). For the union of bisimulations a careful analysis yields *extended cartesian functors* as the least generalisation of polynomial functors (see Section 3.5). For coalgebras of extended cartesian functors the result about the union of bisimulations holds under additional (reasonable) assumptions.

3. Coalgebras for Binary Methods

This chapter presents an approach to define a notion of coalgebra that is capable of modelling arbitrary method types (including binary methods) as they occur in object-oriented programming. A large part of the results described in this chapter appeared originally in (Tews, 2000b), the extended version (Tews, 2001), and in (Tews, 2002b).

In the course of this chapter I present three different generalisations of polynomial functors: *higher-order polynomial functors*, *extended polynomial functors*, and *extended cartesian functors*. They are all functors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$, for a bicartesian closed category $\mathbb{C}$. Higher-order polynomial functors are the most general class. They can model all method types that have been built up from constants, products, coproducts, and exponents. The other two classes restrict the use of the exponent. Thereby it is possible to prove certain properties for the restrictions that do not hold in the general case. For an intuitive description of the differences of all these classes of functors see Table 1.1 in the introduction on page 6 and the explanation there.

The first section of this chapter discusses binary methods and especially the problems they pose in a theoretical approach to object-oriented programming. Section 3.2 defines the class of higher-order polynomial functors (Definition 3.2.1 on page 79) and an associated notion of coalgebra. The following Section 3.3 defines the notions of bisimulation and invariants for such coalgebras. It turns out that bisimulations and invariants for coalgebras of higher-order polynomial functors lack many desired properties in general. It does, for instance, not hold that they are closed under intersection. Section 3.4 defines the first restriction of higher-order polynomial functors: the class of extended polynomial functors (Definition 3.4.1 on page 93). Many familiar results (like the one about intersections) can be proved for extended polynomial functors. So Section 3.4 is a large collection of results about coalgebras of extended polynomial functors. Besides the results that I proved, there is an important deficiency for extended polynomial functors: For coalgebras of extended polynomial functors there is is no greatest bisimulation in general. In Section 3.5 I consider the subclass of extended cartesian functors. For these functors it is possible to adopt the result of (Poll and Zwanenburg, 2001) that greatest bisimulations exist for a suitable strengthened definition of bisimulation. In Section 3.6 I investigate the existence of final coalgebras for generalised polynomial functors. The last section judges the results obtained in this chapter.

3.1. The Problem of Binary Methods

The term *binary method* comes from object-oriented programming and describes methods that take an additional argument of their hosting class. Consider the declaration of a class of (functional) points in a plane:

```
class Point
   methods
      get_x   : Self ⟶ real
      get_y   : Self ⟶ real
      move    : Self × real × real ⟶ Self
      equal   : Self × Self ⟶ bool
end Point
```

As usual in object-oriented programming I use the keyword Self to denote the type of the class that is being defined. In a typical object-oriented programming language all methods get an implicit first argument of type Self that is not mentioned in the type of the methods. On invocation, the object for which the method is called, takes the place of the first argument. Inside the method body this implicit first argument is available via a special keyword: for example this in C++ (Stroustrup, 1997) or Current in Eiffel (Meyer, 1992). In this thesis I always work with the full type of the methods, including the implicit Self–argument like in the preceding example.

In real programming languages such as Eiffel, C++ or OCAML (Leroy et al., 2001) one would implement the operations get_x and get_y as attributes or fields. The intention here is, that they deliver the X and the Y coordinate of the point, respectively. The method move is an ordinary method that changes the internal state of the current object. In most programming languages the method move would be a procedure (or a function of type void) that changes the current object in place and returns nothing. The aim of this thesis is to describe a verification environment for object-oriented programming. For that it is necessary to model objects in a functional way: The method move leaves the current object untouched and returns instead the (changed) successor state. The method equal takes two points and returns true, if the two points are to be considered equal for an outside observer (note, that this might be the case, even if the internal state of the two points differs). The method equal is usually called a binary method for obvious reasons. By a standard abuse of terminology, also n–ary methods, which take more than two arguments of type Self, are referred to as binary methods (Bruce et al., 1995).

A second (standard) example where binary methods are important are classes that implement numbers. All the usual arithmetic operations would be binary methods there. A third example are sets. There the set-theoretical operations union and intersection would be binary methods. Another example connected with side effects in a purely functional setting is discussed below.

In (Tews, 2001) I attempt a further generalisation of the term binary method: Also higher-order methods, where Self occurs in the type of the higher-order argument are called binary methods. The common characteristic of binary methods is, that their types contain at least two occurrences of Self in contravariant position. A precise definition of the term binary method in the context of the specification language CCSL is given in Chapter 4 on page 143.

For an example of these more general binary methods consider the following class NeighbourhoodPoint, which is derived from Point by *inheritance*.

```
class NeighbourhoodPoint
   inherit from Point
   methods
      register_neighbour:      Self × Addr ⟶ Self
      move_with_neighbours: Self × (Addr ⟶ Self) × real × real ⟶ (Addr ⟶ Self)
end NeighbourhoodPoint
```

The idea here is that different points of a plane are stored at addresses of type Addr. A function Addr⟶Self captures the state of the plane and can be used to retrieve single points. With the method register_neighbour one can assign neighbour points to a given point. This assignment is done by addresses so that the neighbours can themselves change their state without affecting the neighbourhood relationship. The method move_with_neighbours can be used to move a point together with all assigned neighbours. This method first needs the state of the whole plane as an argument because it has to access the neighbour points. It further changes the state of the whole plane, therefore it also returns a function Addr⟶Self. This example shows how general binary methods can be used to model side effects in a functional setting.

Note that generalised binary methods are only available in a few programming languages. In Java and Eiffel there is no function type constructor, so one cannot declare an argument of type Addr⟶Self. Also C++ lacks function types but there one can use pointers to functions. Only in OCAML there is no restriction on the type of arguments.

Binary methods do not occur very frequently in practice. Nevertheless they are an important ingredient of object-oriented programming. So any approach to a semantical foundation of object-oriented programming is incomplete, if it does not treat binary methods. However, the usual practice in object-oriented programming makes binary methods a very tough theoretical problem. In order to judge the results of this chapter it is necessary to illustrate these problems in greater detail.

Consider the following subclass ColouredPoint of class Point. For simplicity I model colours with natural numbers:

```
class ColouredPoint
   inherit from Point
   methods
```

```
      colour : Self ⟶ nat
      equal  : Self × Self ⟶ bool
end Point
```

The method equal is listed again to indicate that, in an implementation the method equal should be overridden to take the colour into account, when comparing coloured points. In most object-oriented languages an object of the subclass ColouredPoint can be used where an object of the super class Point is expected. Assume that we pass a coloured point p into a procedure that expects an object of class Point and assume further that this procedure calls the method equal with a second argument q of class Point. In this case the implementation of class ColouredPoint for the method equal would be called.[1] This code would try to access the colour field of q. Depending on the actual language used, this can yield anything between strange results, a runtime exception (possibly caught by the program), and a crash of the whole system. Strictly speaking the problem is not caused by the binary method itself. The origin lies in the fact that in the class ColouredPoint we specialised the second argument of equal to a subtype (namely ColouredPoint) of its original type (Point). The specialisation of argument types during inheritance is an important technique in object-oriented programming. It is often explicitly allowed in programming languages at the price of loosing (static) type safety. One prominent example is Eiffel (Meyer, 1992; Cook, 1989).[2] In contrast the specialisation of a result type of a method to a subtype is harmless.

In the last decades many researchers proposed semantic foundations for object-oriented programming on the basis of type theory. One of the most worked out systems is the ς–calculus of (Abadi and Cardelli, 1996). A number of approaches to encode object-oriented programming in various versions of λ–calculi is compared in (Bruce et al., 1997). In all the work inspired by type theory, one aim is to give a type system that prevents type errors as in the example above. A large number of different proposals about how to solve the typing problem with binary methods is compared in (Bruce et al., 1995). To illustrate the difficulty I sketch two possible solutions in the following.

One solution, which is for instance adopted by OCAML, is to separate subtyping and inheritance following the slogan *Inheritance is not subtyping* from (Cook et al., 1990). In this approach the typechecker would forbid the user to pass a coloured point to a procedure that expects an ordinary point. However, this approach is quite restrictive. It denies many useful applications of argument type specialisation. (In OCAML some —but not all— of these applications can be mimicked through polymorphic classes; see Part II of (Leroy et al., 2001)).

Another interesting solution is proposed by Castagna in (Castagna, 1997). He suggests to enrich the basic λ–calculus with *multimethods* and to use a more intelligent

[1] I assume that late binding applies to the method equal, so in terms of C++, equal should be declared as *virtual*.

[2] Strictly speaking Eiffel is defined as a type safe language, but no compiler currently available implements all the necessary checks (Eiffel FAQ, 2001).

strategy for overriding and method dispatch. In the above example the method equal of class ColouredPoint would have two implementations. The algorithm of dynamic dispatch would take both argument points into account (instead of only the first one) when choosing which implementation should be called. In the above example where the method equal is called for the coloured point p with an ordinary point q as argument, the method equal of class Point would be called. The approach of Castagna is type safe. However, programs compiled with this technique might sometimes execute different methods than the user expects.

One can summarise that the type-theoretic approaches translate terms and types of object-oriented languages into type theory. They do neither describe valid implementations on a general level, nor do they define behavioural equivalence (bisimilarity) of objects. The employed type theories are complex like for instance $F_{\omega,\leq}$ (Pierce and Steffen, 1997) and do not have a set-theoretic semantics (Reynolds, 1984).

A completely different approach to give a semantics to object-oriented languages was proposed by (Reichel, 1995). There the interface of a class is captured by an endofunctor T on a category $\mathbb{C}$ with suitable structure. A class, that is an actual implementation of the interface, is given by a coalgebra $c : X \longrightarrow T(X)$. Here X is the collection of all objects, or more precisely, all possible objects states. Applying the coalgebra to one object yields the results of all methods and the values of all attributes *at once.* It is important to notice, that the coalgebraic approach models object-orientation on a different level and addresses different points than the type-theoretic approach described above. Using coalgebras one gets uniform definitions for terms like behaviour of an object, behavioural equivalence between objects (bisimilarity), and invariance. Additionally one has a suitable definition and proof principle: *Coinduction.* As a semantic universe one can choose an arbitrary category, especially the familiar category **Set** of sets and total functions. The coalgebraic approach does not directly address the problems of inheritance, subtyping, late binding, and overriding. These phenomena have to be modelled separately in the chosen category, see for instance (Jacobs, 1996b; Jacobs, 1996a; Poll, 2001) or Section 4.8 (starting on page 209) in the present thesis.

It comes now as little surprise that binary methods are difficult in coalgebraic specification for completely different reasons than in the type-theoretic approaches. In coalgebraic specification binary methods are difficult, because endofunctors are not expressive enough to model signatures with binary methods. To be precise, a signature with a binary method gives rise to an endofunctor on a category $\mathbb{C}$ only if each arrow in $\mathbb{C}$ is revertible (i.e., for every $f : X \longrightarrow Y$ there must be a $\overline{f} : Y \longrightarrow X$).

This chapter elaborates on one possible solution, namely to use a different class of functors to model signatures. I propose to use the class of *higher-order polynomial functors* (Definition 3.2.1 on page 79) or one of its subclasses *extended polynomial functors* (Definition 3.4.1 on page 93) or *extended cartesian functors* (Definition 3.5.1 on page 108). The difficulty in this approach is, that I have to give new definitions for coalgebra, bisimulation, and invariant for the new classes of functors. The whole standard literature on coalgebras for endofunctors (Rutten, 2000; Gumm, 1999) does not apply to the new notion of coalgebra.

Before I present my solution I discuss two other proposals on how to deal with binary methods in the coalgebraic approach. The first solution is to omit the binary methods from the signature and define them as *definitional extension* (Jacobs, 1996a). This can be done, if the behaviour of the binary method is completely defined in terms of the other methods and attributes. For instance in the point example one usually expects that for all points p and q it holds that

$$\mathsf{equal}(p, q) \qquad \text{if and only if} \qquad \mathsf{get\_x}(p) = \mathsf{get\_x}(q) \quad \text{and} \quad \mathsf{get\_y}(p) = \mathsf{get\_y}(q) \qquad (*)$$

Under this assumption one can remove equal from the signature Point and take instead $(*)$ as a generic definition. The disadvantage of using definitional extensions is immediate: One can only work with binary methods that can be defined in the logic of the specification environment. Further, the observations that can be made through the binary method are determined by the other methods.

Based on similar ideas (Hennicker and Kurz, 1999) present algebraic extensions of coalgebras. There the algebraic extension of a coalgebraic signature contains operations that do not contribute to the observable behaviour of any object (which is ensured by a technical condition). Binary methods with a codomain of Self can be modelled as algebraic extension (so the method equal *does not* fit into an algebraic extension). The advantage of algebraic extensions with respect to definitional extensions is that the requirement of providing a definition is relaxed into checking a condition. However, the behaviour of the binary methods is still determined by the other methods.

The second solution of the problem with binary methods is to shift the focus of the specification from single objects to whole systems of objects. Assume a class signature C with a binary method bm. Any nontrivial application of bm involves several different objects. So its very likely that apart from the class specification itself one also models additional structure to store possibly many objects of the class C. One can circumvent the problems with binary methods if one specifies the whole object system together with the objects at once. This is best illustrated with an example.

Instead of single points we develop a signature for a structure that can store many points. To access the different points we use an index type Index (which I would like to leave open).

```
class PointStore
   methods
      get_x  : Self × Index ⟶ real
      get_y  : Self × Index ⟶ real
      move   : Self × Index × real × real ⟶ Self
      equal  : Self × Index × Index ⟶ bool
end PointStore
```

To invoke a method we need now an object of PointStore *and* an index, which determines on which of the possible many points the method is run. Note that the method equal takes now two arguments of type Index, so the signature of PointStore can be modelled with a polynomial functor.

This approach to circumvent the problem with binary methods is used in the formalisation of Java within the LOOP project (van den Berg et al., 1999; Huisman, 2001). It works well there, because the semantics by Jacobs and colleagues does not preserve object types: All objects of all possible Java classes are mapped to the type ObjectCell that can model any state of any object.

The approach to model the whole system (instead of single classes) gets complicated if several class types should be modelled. A second disadvantage is that the coalgebraic machinery yields a notion of bisimilarity for the whole system, but not for individual objects. Third, in an axiomatic specification one has to give many axioms that state that objects at different indexes are independent. In the example of PointStore one needs for instance

$$\mathsf{get\_x}(\mathsf{move}(s, i, r_1, r_2), j) \quad = \quad \mathsf{get\_x}(s, j)$$

for all objects s of PointStore all indices i and j with $i \neq j$.

3.2. Higher-Order Polynomial Functors

This section introduces the class of higher-order polynomial functors and a generalised notion of coalgebra. Higher-order polynomial functor are a proper superclass of polynomial functors. They form a conservative extension of polynomial functors: every coalgebra for a polynomial functor is also a coalgebra for a higher-order polynomial functor.

Definition 3.2.1 (Higher-order polynomial functors) Assume a bicartesian closed category $\mathbb{C}$. A functor $H : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ is called a *higher-order polynomial functor*, if it is defined as one of the cases

$$H(Y, X) \quad = \quad \begin{cases} X \\ A \\ H_1(Y, X) \times H_2(Y, X) \\ H_1(Y, X) + H_2(Y, X) \\ H_1(X, Y) \Rightarrow H_2(Y, X) \end{cases}$$

where A is an arbitrary object of $\mathbb{C}$ and H_1 and H_2 are previously defined higher-order polynomial functors. The morphism part is defined in the obvious way:

$$H(g, f) \quad = \quad \begin{cases} f \\ \mathrm{id}_A \\ H_1(g, f) \times H_2(g, f) \\ H_1(g, f) + H_2(g, f) \\ H_1(f, g) \Rightarrow H_2(g, f) \end{cases} \qquad \text{if } H(Y, X) \;=\; \begin{cases} X \\ A \\ H_1(Y, X) \times H_2(Y, X) \\ H_1(Y, X) + H_2(Y, X) \\ H_1(X, Y) \Rightarrow H_2(Y, X) \end{cases}$$

Higher-order polynomial functors work on two arguments. This is necessary to separate occurrences of the arguments with positive and negative variance. Both arguments are therefore swapped on the left hand side of $\Rightarrow$.

Definition 3.2.2 (Category of Coalgebras) Let $H : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ be a higher-order polynomial functor. A H–coalgebra is a morphism $c : X \longrightarrow H(X, X)$ in $\mathbb{C}$. Let $d : Y \longrightarrow H(Y, Y)$ be another H–coalgebra. An arrow $f : X \longrightarrow Y$ is a H–coalgebra morphism $f : c \longrightarrow d$, if the following diagram commutes (recall that $H(X, f) = H(\mathrm{id}_X, f)$):

$$\begin{tikzcd}
X \arrow[r, "c"] \arrow[dd, "f"'] & H(X,X) \arrow[dr, "{H(X,f)}"] & \\
 & & H(X,Y) \\
Y \arrow[r, "d"] & H(Y,Y) \arrow[ur, "{H(f,Y)}"'] &
\end{tikzcd}$$

Because H preserves composition and identities the composition of two coalgebra morphisms is a coalgebra morphism again. So the above defines the category $\mathit{CoAlg}(H)$ of H–coalgebras for each higher-order polynomial functor H.

The notion of subcoalgebras is defined as for polynomial functors. Let H be a higher-order polynomial functor on the category **Set** (i.e., $H : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}$). A coalgebra $d : Y \longrightarrow H(Y, Y)$ is a subcoalgebra of $c : X \longrightarrow H(X, X)$ if $Y \subseteq X$ and if the inclusion $\iota : Y \hookrightarrow X$ is a coalgebra morphism $d \xrightarrow{\iota} c$.

Remark 3.2.3 Every polynomial functor is also a higher-order polynomial functor. If a higher-order polynomial functor H is equivalent to a polynomial functor F (i.e. if $H(Y, X) = F(X)$ for all X and Y) then the above pentagon collapses to the square 2.6 in Definition 2.6.1.

Example 3.2.4 Coalgebras of higher-order polynomial functors can be used to model classes of object-oriented languages with arbitrary method types. For the class of neighbourhood points from the introduction one gets the following functor on **Set**.

$$\mathsf{NeighbourhoodPointIface}(Y, X) = \begin{cases} \mathbb{R} \times \mathbb{R} \quad \times \\ (\mathbb{R} \times \mathbb{R} \longrightarrow X) \quad \times \\ (Y \longrightarrow \mathsf{bool}) \quad \times \\ (A \longrightarrow X) \quad \times \\ \Big((A \longrightarrow Y) \times \mathbb{R} \times \mathbb{R} \longrightarrow (A \longrightarrow X)\Big) \end{cases}$$

Here, bool is the set of booleans, $\mathbb{R}$ is used for the set of the real numbers and A is the set of addresses.

A specific class, which realizes the interface of points, corresponds to a coalgebra $c : X \longrightarrow \mathsf{NeighbourhoodPointIface}(X, X)$. We get the single methods as projections:

$$\begin{aligned}
\mathsf{get\_x}_c &= \pi_1 \circ c \\
\mathsf{get\_y}_c &= \pi_2 \circ c \\
\mathsf{move}_c &= \pi_3 \circ c \\
\mathsf{equal}_c &= \pi_4 \circ c \\
\mathsf{register\_neighbour}_c &= \pi_5 \circ c \\
\mathsf{move\_with\_neighbours}_c &= \pi_6 \circ c
\end{aligned}$$

If $d : Y \longrightarrow \mathsf{NeighbourhoodPointIface}(Y, Y)$ is another $\mathsf{NeighbourhoodPointIface}$–coalgebra, then a function $f : X \longrightarrow Y$ is a neighbourhood-point morphism $c \longrightarrow d$ precisely if for all $x, x' \in X$, $r_1, r_2 \in \mathbb{R}$ and $e : A \longrightarrow X$ it holds that[3]

$$\begin{aligned}
\mathsf{get\_x}_c(x) &= \mathsf{get\_x}_d(f(x)) \\
\mathsf{get\_y}_c(x) &= \mathsf{get\_y}_d(f(x)) \\
f\big(\mathsf{move}_c(x, r_1, r_2)\big) &= \mathsf{move}_d(f(x), r_1, r_2) \\
\mathsf{equal}_c(x, x') &= \mathsf{equal}_d(f(x), f(x')) \\
f\big(\mathsf{register\_neighbour}_c(x, a)\big) &= \mathsf{register\_neighbour}_d(f(x), a) \\
\lambda a \,.\, f\big(\mathsf{move\_with\_neighbours}_c(x, e, r_1, r_2)(a)\big) &= \\
&\qquad \mathsf{move\_with\_neighbours}_d\big(f(x),\ (\lambda a \,.\, f(e(a))),\ r_1, r_2\big) \qquad \blacksquare
\end{aligned}$$

Proposition 2.6.2 (on page 55) can be extended to higher-order polynomial functors. Note, that a natural transformation $\eta : H \Longrightarrow K$ between two higher-order polynomial functors $H, K : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ is a collection of morphisms indexed by pairs: $\eta_{Y,X} : H(Y, X) \longrightarrow K(Y, X)$. Its defining property is the commutation of the following diagram for any two morphisms f and g:

$$\begin{tikzcd}
Y & X \arrow[d, "f"] & H(Y,X) \arrow[r, "\eta_{Y,X}"] \arrow[d, "H(g,f)"'] & K(Y,X) \arrow[d, "K(g,f)"] \\
V \arrow[u, "g"] & U & H(V,U) \arrow[r, "\eta_{V,U}"] & K(V,U)
\end{tikzcd}$$

Lemma 3.2.5 *Let H and K be two higher-order polynomial functor. A natural transformation $\eta : H \Longrightarrow K$ gives rise to a functor $CoAlg(H) \longrightarrow CoAlg(K)$ by postcomposition.*

[3] In the following equations I use currying and write $move(x, r_1, r_2)$ instead of $move(x)(r_1, r_2)$ to get a better presentation.

Proof As in the proof of Proposition 2.6.2 the main obligation is to show that H-coalgebra morphisms are mapped to K coalgebra morphisms. The situation is as follows

$$\begin{tikzcd}
X \arrow[r, "c"] \arrow[dd, "f"'] & H(X,X) \arrow[rr, "\eta_{X,X}"] \arrow[dr, "H(X,f)"] & & K(X,X) \arrow[dr, "K(X,f)"] & \\
& (1) & H(X,Y) \arrow[rr, "\eta_{X,Y}"] \arrow[ur, phantom, "(2)"] \arrow[dr, phantom, "(3)"] & & K(X,Y) \\
Y \arrow[r, "d"] & H(Y,Y) \arrow[rr, "\eta_{Y,Y}"] \arrow[ur, "H(f,Y)"] & & K(Y,Y) \arrow[ur, "K(f,Y)"] &
\end{tikzcd}$$

Parts (2) and (3) commute by the naturality of η, part (1) commutes because f is a H–coalgebra morphism. Thus, the outer pentagon commutes. □

3.3. Invariants and Bisimulations for Higher-Order Polynomial Functors

Bisimulations are used in various process calculi (for example in (Milner, 1989)) and in the field of coalgebras to capture behavioural equivalence. *Invariants* are predicates on the state space of a coalgebra that are maintained by the coalgebra. That is, starting from a state in the invariant, all successor states that can be obtained via the coalgebra are in the invariant predicate again. As explained in Section 2.6 there exist two approaches to define bisimulations and invariants. I first follow the Hermida/Jacobs approach and extend predicate and relation lifting from (Hermida and Jacobs, 1998) to higher-order polynomial functors. At the end of this section I compare the notion of Aczel/Mendler bisimulation with Hermida/Jacobs bisimulation. It turns out that both approaches yield different notions of bisimulation and invariant. The Aczel/Mendler approach yields a definition of bisimulation that does not capture the intuitive notion of behavioural equivalence: It is possible to construct an Aczel/Mendler bisimulation R, which relates two states, such that their successor states are not related by R and may give different observations (see Example 3.3.11 below).

Consider a fibration $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$. Predicate and relation lifting take a higher-order polynomial functor on the base category $H : \mathbb{B}^{\mathrm{op}} \times \mathbb{B} \longrightarrow \mathbb{B}$ and transform it into a functor on the total category $\mathrm{Pred}(H) : \mathbb{E}^{\mathrm{op}} \times \mathbb{E} \longrightarrow \mathbb{E}$ (or $\mathrm{Rel}(H) : \mathbf{Rel}(\mathbb{E})^{\mathrm{op}} \times \mathbf{Rel}(\mathbb{E}) \longrightarrow \mathbf{Rel}(\mathbb{E})$ for relation lifting, where $\mathbf{Rel}(\mathbb{E})$ is the category of relations obtained from $\begin{array}{c}\mathbb{E}\\ \downarrow p\\ \mathbb{B}\end{array}$). This new functor on the total category is obtained by simply substituting in H the bicartesian structure of the total category for the bicartesian structure of the base. For constants one uses truth (for predicate lifting) and equality (for relation lifting). For a polynomial functor F the predicate and the relation lifting of F is (co-)fibred over F under additional assumptions on $\mathbb{B}$ and on $\mathbb{E}$. This is no longer the case for higher-order polynomial functors.

For the reasons explained in the beginning of Section 2.4 I switch now to the concrete fibrations $\begin{array}{c}\mathbf{Pred}\\ \downarrow\\ \mathbf{Set}\end{array}$ and $\begin{array}{c}\mathbf{Rel}\\ \downarrow\\ \mathbf{Set}\end{array}$. Functors that appear in the following are functors on the category **Set**, if not otherwise mentioned.

Definition 3.3.1 (Predicate and Relation lifting) Let $Q \subseteq Y$ and $P \subseteq X$ be two predicates, $S \subseteq V \times Y$ and $R \subseteq U \times X$ two relations, and H a higher-order polynomial functor. Its predicate lifting $\mathrm{Pred}(H)(Q,P) \subseteq H(Y,X)$ and its relation lifting $\mathrm{Rel}(H)(S,R) \subseteq H(V,U) \times H(Y,X)$ are defined by induction on the structure of H:

- If $H(Y,X) = X$, then

$$\begin{array}{rcl} \mathrm{Pred}(H)(Q,P) & = & P \\ \mathrm{Rel}(H)(S,R) & = & R \end{array}$$

- If $H(Y,X) = A$ (for A a constant set), then

$$\begin{array}{rcccl} \mathrm{Pred}(H)(Q,P) & = & \top_A & = & (A \subseteq A) \\ \mathrm{Rel}(H)(S,R) & = & \mathrm{Eq}(A) & = & \big(\{(a,a) \,|\, a \in A\} \subseteq A \times A\big) \end{array}$$

- If $H(Y,X) = H_1(Y,X) \times H_2(Y,X)$, then

$$\begin{array}{rcl} \mathrm{Pred}(H)(Q,P) & = & \mathrm{Pred}(H_1)(Q,P) \times_{\mathrm{P}} \mathrm{Pred}(H_2)(Q,P) \\ & = & \big\{(x,y) \mid \mathrm{Pred}(H_1)(Q,P)(x) \;\text{ and }\; \mathrm{Pred}(H_2)(Q,P)(y)\big\} \\ \mathrm{Rel}(H)(S,R) & = & \mathrm{Rel}(H_1)(S,R) \times_{\mathrm{R}} \mathrm{Rel}(H_2)(S,R) \\ & = & \big\{((x_1,x_2),(y_1,y_2)) \mid \mathrm{Rel}(H_1)(S,R)(x_1,y_1) \\ & & \qquad\qquad \text{and } \; \mathrm{Rel}(H_2)(S,R)(x_2,y_2)\big\} \end{array}$$

- If $H(Y,X) = H_1(Y,X) + H_2(Y,X)$, then

$$\begin{array}{rcl} \mathrm{Pred}(H)(Q,P) & = & \mathrm{Pred}(H_1)(Q,P) +_{\mathrm{P}} \mathrm{Pred}(H_2)(Q,P) \\ & = & \big\{\kappa_1\, x \mid \mathrm{Pred}(H_1)(Q,P)(x)\big\} \\ & & \quad \cup \; \big\{\kappa_2\, y \mid \mathrm{Pred}(H_2)(Q,P)(y)\big\} \\ \mathrm{Rel}(H)(S,R) & = & \mathrm{Rel}(H_1)(S,R) +_{\mathrm{R}} \mathrm{Rel}(H_2)(S,R) \\ & = & \big\{(\kappa_1\, x_1, \kappa_1\, y_1) \mid \mathrm{Rel}(H_1)(S,R)(x_1,y_1)\big\} \\ & & \quad \cup \; \big\{(\kappa_2\, x_2, \kappa_2\, y_2) \mid \mathrm{Rel}(H_2)(S,R)(x_2,y_2)\big\} \end{array}$$

- If $H(Y,X) = H_1(X,Y) \Rightarrow H_2(Y,X)$, then

$$\begin{array}{rcl}
\mathrm{Pred}(H)(Q,P) & = & \mathrm{Pred}(H_1)(P,Q) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(H_2)(Q,P) \\
& = & \big\{ f \mid f : H_1(X,Y) \longrightarrow H_2(Y,X) \text{ such that} \\
& & \quad \forall a \in H_1(X,Y) \,.\, \mathrm{Pred}(H_1)(P,Q)(a) \text{ implies } \mathrm{Pred}(H_2)(Q,P)(f(a)) \big\} \\
\mathrm{Rel}(H)(S,R) & = & \mathrm{Rel}(H_1)(R,S) \Rightarrow_{\mathrm{R}} \mathrm{Rel}(H_2)(S,R) \\
& = & \big\{ (f,g) \mid f : H_1(U,V) \longrightarrow H_2(V,U),\, g : H_1(X,Y) \longrightarrow H_2(Y,X) \\
& & \quad \text{such that } \forall a \in H_1(U,V), b \in H_1(X,Y) \,.\, \mathrm{Rel}(H_1)(R,S)(a,b) \\
& & \qquad \text{implies } \mathrm{Rel}(H_2)(S,R)(f(a),g(b)) \big\}
\end{array}$$

Predicate and relation lifting for higher-order polynomial functors works on two arguments. In $\mathrm{Pred}(H)(Q,P)$ (respectively $\mathrm{Rel}(H)(S,R)$) the first argument Q (respectively S) is used for the contravariant occurrences of Y in $H(Y,X)$. The second argument P (respectively R) is for the covariant argument of H. The effect is the following: An element $t \in H(Y,X)$ is in $\mathrm{Pred}(H)(Q,P)$ if Q and P hold pointwise on the contravariant and covariant positions in t. For an additional $s \in H(V,U)$ we have $\mathrm{Rel}(H)(S,R)(s,t)$ only if s and t have the same structure. This means, that if $H = H_1 + H_2$, then s and t come either both from the first component or both from the second one. Further $\mathrm{Rel}(H)(S,R)$ requires S, R and equality to hold pointwise for the contravariant argument, covariant argument and constants, respectively.

This definition of predicate and relation lifting is a conservative extension of Definition 2.6.3 (on page 58). If H is equivalent to a polynomial functor F then, in the liftings $\mathrm{Pred}(H)(Q,P)$ and $\mathrm{Rel}(H)(S,R)$, the first arguments Q and S are never used. Therefore, in this case, $\mathrm{Pred}(H)(Q,P) = \mathrm{Pred}(F)(P)$ and $\mathrm{Rel}(H)(S,R) = \mathrm{Rel}(F)(R)$.

I need the following result below.

Lemma 3.3.2 *Let H be a higher-order polynomial functor.*

1. *Predicate and relation lifting are monotone, that is, the definitions of* $\mathrm{Pred}(H)$ *and* $\mathrm{Rel}(H)$ *extend to functors: For suitable predicates Q, Q', P, P' and relations S, S', R, R' we have*

$$\begin{array}{ccccc}
Q \subseteq Q' & \textit{and} & P \subseteq P' & \textit{implies} & \mathrm{Pred}(H)(Q',P) \subseteq \mathrm{Pred}(H)(Q,P') \\
S \subseteq S' & \textit{and} & R \subseteq R' & \textit{implies} & \mathrm{Rel}(H)(S',R) \subseteq \mathrm{Rel}(H)(S,R')
\end{array}$$

2. *Predicate lifting commutes with truth:*

$$\mathrm{Pred}(H)(\top_Y, \top_X) \quad = \quad \top_{H(Y,X)}$$

3. *Relation lifting commutes with equality:*

$$\mathrm{Rel}(H)(\mathrm{Eq}(X), \mathrm{Eq}(Y)) \quad = \quad \mathrm{Eq}(H(X,Y))$$

4. *For two relations $S \subseteq U \times V$ and $R \subseteq X \times Y$ relation lifting commutes with $(-)^{\mathrm{op}}$ in the following sense:*

$$\mathrm{Rel}(H)(S^{\mathrm{op}}, R^{\mathrm{op}}) \quad = \quad (\,\mathrm{Rel}(H)(S, R)\,)^{\mathrm{op}}$$

Proof By induction on the structure of H using Lemmas 2.4.5, 2.4.7, 2.4.8, and 2.4.11. □

It is impossible to obtain similar results for the commutation of predicate and relation lifting with $\wedge, \vee$ or relational composition. This fails because it is not possible to obtain equations instead of the subset relations in the Lemmas 2.4.9, 2.4.10, and 2.4.12. The lifting of higher-order polynomial functors is also not fibred, because this would require the lifting to be cofibred, which fails for the exponent (see Example 2.4.16).

Definition 3.3.3 (Hermida/Jacobs Invariant and Bisimulation)
Let H be a higher-order polynomial functor and $c : X \longrightarrow H(X, X)$ and $d : Y \longrightarrow H(Y, Y)$ be two H–coalgebras.

- A predicate $P \subseteq X$ is a *Hermida/Jacobs H–invariant* for c if for all $x \in X$

$$P(x) \quad \text{implies} \quad \mathrm{Pred}(H)(P, P)(c(x))$$

- A relation $R \subseteq X \times Y$ is a *Hermida/Jacobs H–bisimulation* for c and d if for all $x \in X, y \in Y$

$$R(x, y) \quad \text{implies} \quad \mathrm{Rel}(H)(R, R)(c(x), d(y))$$

Note that one could equivalently formulate the definition as follows: Let $\widehat{H}$ be the functor defined as $\widehat{H}(Y, X) = Y \Rightarrow H(Y, X)$, then P is a H–invariant for c if $\mathrm{Pred}(\widehat{H})(P, P)(c)$. And R is a H–bisimulation for c and d if $\mathrm{Rel}(\widehat{H})(R, R)(c, d)$ holds. Another variation to express the same thing uses the substitution functor for the coalgebra c: The predicate P is an invariant if $P \subseteq c^* \,\mathrm{Pred}(H)(P, P)$ holds. Similarly for bisimulations.

The preceding definition is a straightforward generalisation of Definition 2.6.5 (on page 60). We will see in the following that the notion of Hermida/Jacobs invariant is not optimal both with respect to its properties and with respect to the intuition about invariance. Therefore in the context of CCSL the slightly different notion of *strong invariant* is used. Strong invariants are discussed in Section 3.4.6 (starting on page 103).

In the following I analyse the properties of Hermida/Jacobs invariants and bisimulations for higher-order polynomial functors. The term invariant (without qualification) denotes always a Hermida/Jacobs invariant from the preceding definition, similarly the term bisimulation abbreviates Hermida/Jacobs bisimulation.

Example 3.3.4 As example I describe invariants and bisimulations for neighbourhood points. Let

$$c : X \longrightarrow \mathsf{NeighbourhoodPointIface}(X, X)$$

and

$$d : Y \longrightarrow \mathsf{NeighbourhoodPointIface}(Y, Y)$$

be two point coalgebras. A predicate $P \subseteq X$ is an invariant for c if $P(x)$ implies all of the following points.

- $\forall r_1, r_2 \in \mathbb{R} \,.\, P\big(\mathsf{move}_c(x, r_1, r_2)\big)$
- $\forall a \in A \,.\, P\big(\mathsf{register\_neighbour}_c(x, a)\big)$
- if e is a plane (i.e., a function $A \longrightarrow X$) such that $\forall a \in A \,.\, P(e(a))$ then it must hold that $\forall r_1, r_2 \in \mathbb{R} \,.\, \forall a \in A \,.\, P\big(\mathsf{move\_with\_neighbours}(x, e, r_1, r_2)\ (a)\big)$

Note that a successor state of the move_with_neighbours is not required to be in the invariant P if only one point that can be reached via the second argument e is outside of P. This is because in the definition of invariant the co– and the contravariant arguments of predicate lifting are instantiated with the same predicate. Depending on the actual application it might better fit the intuition if, regardless of the argument e, all successor states of the move_with_neighbours method are required to be within P. The notion of *strong invariant* fulfils this latter criterion (see Section 3.4.6 below).

Let us now look under which condition a relation $R \subseteq X \times Y$ forms a bisimulation for neighbourhood points. First it is necessary to lift a relation to the type of planes, which are the second argument of the move_with_neighbour method: Let me call two planes $e_1 : A \longrightarrow X$ and $e_2 : A \longrightarrow Y$ R–related, if for all $a \in A$ it holds that $R(e_1(a), e_2(a))$.[4] Now the relation $R \subseteq X \times Y$ is a bisimulation for neighbourhood points if for all $x \in X$ and $y \in Y$ with $R(x, y)$ all of the following items hold:

- $\mathsf{get\_x}_c(x) \ =_{\mathbb{R}}\ \mathsf{get\_x}_d(y)$
- $\mathsf{get\_y}_c(x) \ =_{\mathbb{R}}\ \mathsf{get\_y}_d(y)$
- $\forall r_1, r_2 \in \mathbb{R} \,.\, R\big(\mathsf{move}_c(x, r_1, r_2),\ \mathsf{move}_d(y, r_1, r_2)\big)$
- $\forall x' \in X, y' \in Y \,.\, R(x', y')$ implies $\mathsf{equal}_c(x, x') \ =_{\mathsf{bool}}\ \mathsf{equal}_d(y, y')$
- $\forall a \in A \,.\, R\big(\mathsf{register\_neighbour}_c(x, a),\ \mathsf{register\_neighbour}_d(y, a)\big)$

[4] The informal notion of $(-)$–related functions makes relation lifting readable for higher-order polynomial functors in this and in following examples. This informal notion is always an abbreviation for the relation lifting of an ingredient functor in the current context. Here, two planes are R–related precisely if they are related by $\mathrm{Rel}(A \Rightarrow X)(R, R) = \{(e_1, e_2) \mid \forall a : A \,.\, R(e_1(a), e_2(a))\}$

- For all R–related planes $e_1 : A \longrightarrow X$ and $e_2 : A \longrightarrow Y$ and all $r_1, r_2 \in \mathbb{R}$ also the two results $\mathsf{move\_with\_neighbours}_c(x, e_1, r_1, r_2)$ and $\mathsf{move\_with\_neighbours}_d(y, e_2, r_1, r_2)$ are R–related. ■

Remark 3.3.5 Sometimes the powerset functor is defined as $\mathbb{P}(X) = X \Rightarrow \mathsf{bool}$. This way the powerset functor could be considered as a higher-order polynomial functor. However the notions of bisimulation and invariant that one gets for the functor $X \Rightarrow \mathsf{bool}$ are not appropriate to model nondeterminism. As liftings one gets

$$\begin{array}{rcl} \mathrm{Pred}(X \longmapsto X \Rightarrow \mathsf{bool})(P) & = & \top_{X \Rightarrow \mathsf{bool}} \\ \mathrm{Rel}(X \longmapsto X \Rightarrow \mathsf{bool})(R) & = & \{(f, g) \mid x\,R\,y \ \text{ implies } \ f\,x = g\,x\} \end{array}$$

So for a coalgebra $c : X \longrightarrow X \Rightarrow \mathsf{bool}$ every predicate is an invariant. And a relation R is a bisimulation if for all x, x', y, and y' it holds that

$$x\,R\,y \wedge x'\,R\,y' \qquad \text{implies} \qquad x' \in c(x) \ \text{ if and only if } \ y' \in c(y)$$

(here I consider $c(x)$ as the set $\{x' \mid c(x)(x') = \top\}$). Note that the Aczel/Mendler approach yields the same result, because Proposition 3.4.20 (on page 102 below) applies to the functor $X \Rightarrow \mathsf{bool}$. The problem is that $X \Rightarrow \mathsf{bool}$ gives the *contravariant* powerset functor, but in order to model nondeterminism one needs the *covariant* powerset functor (compare Section 3.6 of (Rutten, 2000)).

To model nondeterminism (Jacobs, 1995) suggests the following liftings:

$$\begin{array}{rcl} \mathrm{Pred}(\mathbb{P})(P) & = & \{Q \subseteq X \mid Q \subseteq P\} \\ \mathrm{Rel}(\mathbb{P})(R) & = & \{(Q \subseteq X, Q' \subseteq Y) \mid (\forall x \in Q \,.\, \exists y \in Q' \,.\, x\,R\,y) \ \wedge \\ & & \qquad (\forall y \in Q' \,.\, \exists x \in Q \,.\, x\,R\,y) \end{array}$$

These liftings behave in the expected way.

It is now the question whether bisimulations for higher-order polynomial functors enjoy similar properties like bisimulations for weak pullback preserving endofunctors do. For the more general case of higher-order polynomial functors, the possibility to specify arbitrary functional arguments and the contravariant nature of the function type cause problems. So there are only a few positive results.

Proposition 3.3.6 *Let H be a higher-order polynomial functor and $c : X \longrightarrow H(X, X)$ and $d : Y \longrightarrow H(Y, Y)$ be two H–coalgebras.*

1. *The truth predicate $\top_X$ is an invariant for c.*
2. *The equality relation $\mathrm{Eq}(X)$ is a bisimulation for c.*
3. *If $R \subseteq X \times Y$ is a bisimulation for c and d, then R^{op} is a bisimulation for d and c.*

Proof Apply Lemma 3.3.2. □

Fact 3.3.7 None *of the following points hold in general for coalgebras of higher-order polynomial functors.*

1. *bisimulations and invariants are closed under union and intersection,*
2. *the composition of two bisimulations is a bisimulation,*
3. *the graph of a morphism is a bisimulation,*
4. *the image $\coprod_f \top$ of a morphism f is an invariant,*
5. *invariants correspond to subcoalgebras,*
6. *the relation $\coprod_\delta P$ is a bisimulation for an invariant P,*
7. *the predicate $\coprod_{\pi_1} R$ is an invariant for a bisimulation R,*
8. *the relation ${\pi_1}^* P \wedge R$ is a bisimulation for an invariant P and a bisimulation R,*
9. *and finally the kernel of a morphism is a bisimulation.*

For all these facts I constructed counterexamples in the PVS formalisation of this chapter. Here I include only some of them: See the following Example 3.3.8 and Example 3.3.9 for Item 1, Example 3.4.17 (on page 101) for Item 4, Example 3.4.19 (on page 101) for Item 6, Example 3.4.12 (on page 99) for Item 7, and Example 3.4.15 (on page 100) for Item 8. It is worth remarking that the examples for Items (2)–(9) involve the same functor T of the following example. For the union of bisimulations the functor $(Y, X) \longmapsto Y \Rightarrow \mathsf{bool}$ suffices, for the union of invariants one needs $(Y, X) \longmapsto Y \Rightarrow X$.

Example 3.3.8 This example shows two bisimulations such that their intersection is not a bisimulation. Consider the higher-order polynomial functor

$$T(Y, X) \quad \stackrel{\text{def}}{=} \quad (X \Rightarrow Y) \Rightarrow X$$

For two functions $f : U \longrightarrow X$ and $g : Y \longrightarrow V$ its morphism part is

$$\begin{array}{rcl} T(g, f) & : & T(V, U) \longrightarrow T(Y, X) \\ & = & (f \Rightarrow g) \Rightarrow f \\ h : (U \Rightarrow V) \longrightarrow U & \longmapsto & \lambda k : X \longrightarrow Y \,.\, (f \circ h)(g \circ k \circ f) \end{array}$$

To describe the relation lifting for T it is useful to define the auxiliary notion of (R, S)–related functions[5] : For two relations $R \subseteq X \times Y$, $S \subseteq U \times V$ the functions $a : X \longrightarrow U$

[5] Two functions are (R, S)–related if they are related by $\mathrm{Rel}(Y \Rightarrow X)(R, S)$.

and $b : Y \longrightarrow V$ are (R,S)–related, if for all $x \in X$ and $y \in Y$ it holds that $R(x,y)$ implies $S(a(x), b(y))$.

Assuming again two relations $S \subseteq U \times V$ and $R \subseteq X \times Y$ the relation lifting for T is then

$$\begin{array}{rcl} \mathrm{Rel}(T)(S,R) & \subseteq & T(U,X) \times T(V,Y) \\ & = & \big\{(f,g) \mid f : (X \Rightarrow U) \longrightarrow X \,, g : (Y \Rightarrow V) \longrightarrow Y \text{ such that} \\ & & \quad \text{for all } (R,S)\text{–related functions } a \text{ and } b : R\big(f(a), g(b)\big) \;\big\} \end{array}$$

The functor T is trivial in the sense that it does not allow any observations. Therefore there exists a final T–coalgebra (given by the only function $\mathbf{1} \longrightarrow T(\mathbf{1})$) and for any T–coalgebra $Z \longrightarrow T(Z)$ the universal relation $Z \times Z$ is the greatest bisimulation. However, the argument below (and in all examples that use the functor T) applies also to nontrivial higher-order polynomial functors like $T'(Y,X) = (X \Rightarrow Y) \Rightarrow (X \times A)$.

Take now two sets $A \overset{\mathrm{def}}{=} \{a_1, a_2, a_3, a_4\}$, $B \overset{\mathrm{def}}{=} \{b_1, b_2, b_3, b_4\}$ and the relations $R \overset{\mathrm{def}}{=} \{(a_1, b_1), (a_2, b_2)\}$ and $S \overset{\mathrm{def}}{=} \{(a_1, b_1), (a_3, b_3)\}$. Define the following functions:

$$\begin{array}{rcl} f & : & A \longrightarrow A \\ & = & \lambda a : A \,.\, \mathsf{if}\ a = a_1\ \mathsf{then}\ a_1\ \mathsf{else}\ a_4\ \mathsf{endif} \\[1ex] g & : & B \longrightarrow B \\ & = & \lambda b : B \,.\, \mathsf{if}\ b = b_1\ \mathsf{then}\ b_1\ \mathsf{else}\ b_4\ \mathsf{endif} \\[1ex] c & : & A \longrightarrow (A \Rightarrow A) \Rightarrow A \\ & = & \lambda a : A \,.\, \lambda h : A \longrightarrow A \,.\, \mathsf{if}\ h = f\ \mathsf{then}\ a_4\ \mathsf{else}\ a_1\ \mathsf{endif} \\[1ex] d & : & B \longrightarrow (B \Rightarrow B) \Rightarrow B \\ & = & \lambda b : B \,.\, \lambda k : B \longrightarrow B \,.\, \mathsf{if}\ k = g\ \mathsf{then}\ b_4\ \mathsf{else}\ b_1\ \mathsf{endif} \end{array}$$

Obviously, the functions f and g are neither (R,R)–related nor (S,S) related. However, they are $(R \cap S, R \cap S)$–related and $(c(a_1)(f),\ d(b_1)(g)) \notin R \cap S$. That is why R and S are bisimulations for c and d, but $R \cap S$ is not.

The first projections of R and S, the predicates $\coprod_{\pi_1} R = \{a_1, a_2\}$ and $\coprod_{\pi_1} S = \{a_1, a_3\}$ are both invariants for c, but there intersection is not. ■

Example 3.3.9 In this example I consider the functor

$$K(Y,X) \quad \overset{\mathrm{def}}{=} \quad Y \Rightarrow \mathsf{bool}$$

where $\mathsf{bool} \overset{\mathrm{def}}{=} \{\bot, \top\}$ is the set of booleans. For this functor I construct a K–coalgebra for which the union of two bisimulation is not a bisimulation and for which there is also no greatest bisimulation. For a function $g : Y \longrightarrow V$ (the covariant argument position is ignored) the morphism part of K is

$$\begin{array}{rcl} K(g, X) & : & K(V, X) \longrightarrow K(Y, X) \\ & = & (g \Rightarrow \mathrm{id}_{\mathsf{bool}}) \\ h : V \longrightarrow \mathsf{bool} & \longmapsto & h \circ g \end{array}$$

Assume two relations $S \subseteq U \times V$ and $R \subseteq X \times Y$. The relation lifting for K is

$$\begin{array}{rcl} \mathrm{Rel}(K)(S, R) & \subseteq & K(U, X) \times K(V, Y) \\ & = & \big\{(f, g) \mid f : U \longrightarrow \mathsf{bool}, g : V \longrightarrow \mathsf{bool} \ \text{ such that} \\ & & \quad \forall u \in U, v \in V \,.\, S(u, v) \ \text{ implies } \ f(u) = g(v)\big\} \end{array}$$

Define the following coalgebra on the state space $A = \{a_1, a_2\}$.

$$\begin{array}{rcl} c & : & A \longrightarrow K(A, A) \\ & = & \lambda x : A \,.\, \lambda y : A \,.\, \mathsf{if}\ x = y\ \mathsf{then}\ \bot\ \mathsf{else}\ \top\ \mathsf{endif} \end{array}$$

Define the relation $R \subseteq A \times A \overset{\text{def}}{=} \{(a_1, a_2), (a_2, a_1)\}$. It is easy to check that R is a K–bisimulation for c. Also the equality relation $\mathrm{Eq}(A) = \{(a_1, a_1), (a_2, a_2)\}$ is a bisimulation for c. However, the union $R \cup \mathrm{Eq}(A)$ is not. Both bisimulations are maximal: if S is another K–bisimulation for c then either $S \subseteq R$ or $S \subseteq \mathrm{Eq}(A)$. So there is no greatest bisimulation for c.

Note that R is on the edge of (if not behind) an intuitive notion of behavioural equivalence. The Sections 3.4.7 and 3.5 consider stronger notions of bisimulation (excluding R) that behave more nicely with respect to union. ■

In the remainder of this subsection I define invariants and bisimulations, following the approach of Aczel and Mendler. It will turn out, that this gives different notions of invariance and bisimulation (compared to Definition 3.3.3).

Definition 3.3.10 (Aczel/Mendler bisimulations and invariants)
Let $c : X \longrightarrow H(X, X)$ and $d : Y \longrightarrow H(Y, Y)$ be coalgebras for a higher-order polynomial functor H.

- A predicate $P \subseteq X$ is called an *Aczel/Mendler invariant* (for c) if there exists a subcoalgebra on P, that is if there is a coalgebra $p : P \longrightarrow H(P, P)$ such that the inclusion $\iota : P \longrightarrow X$ is a H–coalgebra morphism $p \longrightarrow c$ (i.e., the right diagram below commutes).

- A relation $R \subseteq X \times Y$ is called an *Aczel/Mendler bisimulation* (for c and d) if there exists a coalgebra $r : R \longrightarrow H(R, R)$ such that the projections $\pi_1 : R \longrightarrow X$ and $\pi_2 : R \longrightarrow Y$ are H–coalgebra morphisms (i.e., if the left diagram below commutes).

$$\begin{tikzcd}
X \arrow[r, "c"] & H(X,X) \arrow[dr, "H(\pi_1,X)"] & \\
 & & H(R,X) \\
R \arrow[uu, "\pi_1"] \arrow[dd, "\pi_2"'] \arrow[r, "r"] & H(R,R) \arrow[ur, "H(R,\pi_1)"'] \arrow[dr, "H(R,\pi_2)"] & \\
 & & H(R,Y) \\
Y \arrow[r, "d"] & H(Y,Y) \arrow[ur, "H(\pi_2,Y)"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
P \arrow[r, "p"] \arrow[dd, "\iota"'] & H(P,P) \arrow[dr, "H(X,\iota)"] & \\
 & & H(X,P) \\
X \arrow[r, "c"] & H(X,X) \arrow[ur, "H(\iota,X)"'] &
\end{tikzcd}$$

Example 3.3.11 This examples presents a relation that is an Aczel/Mendler bisimulation but not a Hermida/Jacobs bisimulation. Consider the functor T from Example 3.3.8. Take for a concrete state space the set $M \stackrel{\text{def}}{=} \{m_1, m_2\}$ and let R be the relation that relates only m_1 with itself: $R \stackrel{\text{def}}{=} \{(m_1, m_1)\}$. Define the following T–coalgebras

$$\begin{aligned}
c \quad &: \quad M \longrightarrow (M \Rightarrow M) \Rightarrow M \\
&= \quad \lambda x : M \,.\, \lambda a : M \longrightarrow M \,.\, \mathsf{if}\ a = \mathrm{id}_M\ \mathsf{then}\ m_2\ \mathsf{else}\ m_1\ \mathsf{endif} \\
r \quad &: \quad R \longrightarrow (R \Rightarrow R) \Rightarrow R \\
&= \quad \lambda r : R \,.\, \lambda r' : R \longrightarrow R \,.\, (m_1, m_1)
\end{aligned}$$

The question is now whether R is a bisimulation for c (to instantiate Definition 3.3.10 I take $X = Y = M$ and $c = d$). Intuitively we should expect, that R is not a T–bisimulation for c, because for the state m_1 we get $c(m_1)(\mathrm{id}_M) = m_2$. So if $(m_1, m_1) \in R$ then R should also contain the pair (m_2, m_2), because a bisimulation should be closed under taking successor states. And indeed, since id_M is (R, R)–related with itself, we find that $(c(m_1), c(m_1)) \notin \mathrm{Rel}(T)(R, R)$ and R is not a Hermida/Jacobs bisimulation.

Checking for the Aczel/Mendler bisimulation we find, that

$$\begin{aligned}
T(\pi_1, M) \circ c \quad &: \quad M \longrightarrow (M \Rightarrow R) \Rightarrow M \\
&= \quad \lambda x : M \,.\, \lambda a : M \longrightarrow R \,.\, m_1
\end{aligned}$$

and r is indeed a coalgebra fulfilling the condition of Definition 3.3.10.

This shows that the Aczel/Mendler definition for bisimulation does not capture the basic intuition about bisimulations: it is possible to relate states, which are not behaviourally equivalent. ■

Example 3.3.12 This is an example for the converse situation: I present a relation that is a Hermida/Jacobs bisimulation but not an Aczel/Mendler bisimulation. Consider again the functor T from Example 3.3.8. This time take $M \overset{\text{def}}{=} \{m_1, m_2, m_3, m_4, m_5\}$ and consider the relation $R \overset{\text{def}}{=} \{(m_1, m_1), (m_1, m_2), (m_3, m_4)\}$. Define the following functions:

$$\begin{array}{rcl}
f & : & M \longrightarrow R \\
 & = & \lambda x : M \,.\, \mathsf{if}\ x = m_1\ \mathsf{then}\ (m_3, m_4)\ \mathsf{else}\ (m_1, m_1)\ \mathsf{endif} \\[1ex]
\pi_2 \circ f & : & M \longrightarrow M \\
 & = & \lambda x : M \,.\, \mathsf{if}\ x = m_1\ \mathsf{then}\ m_4\ \mathsf{else}\ m_1\ \mathsf{endif} \\[1ex]
c & : & M \longrightarrow (M \Rightarrow M) \Rightarrow M \\
 & = & \lambda x : M \,.\, \lambda a : M \longrightarrow M \,.\, m_1 \\[1ex]
d & : & M \longrightarrow (M \Rightarrow M) \Rightarrow M \\
 & = & \lambda x : M \,.\, \lambda a : M \longrightarrow M \,.\, \mathsf{if}\ a = \pi_2 \circ f\ \mathsf{then}\ m_5\ \mathsf{else}\ m_1\ \mathsf{endif}
\end{array}$$

Consider the composition $\pi_2 \circ f$. There is no function $g : M \longrightarrow M$ such that g is (R, R)–related to $\pi_2 \circ f$, because this would require both $g\, m_1 = m_3$ and $g\, m_1 = m_1$.

The question is again, if we should consider R as a bisimulation for the coalgebras c and d. Both coalgebras are clearly different on input $\pi_2 \circ f$, but because there is no (R, R)–related function to $\pi_2 \circ f$, we find that $(c(x), d(y)) \in \mathrm{Rel}(T)(R, R)$ for all $(x, y) \in R$. So R is a Hermida/Jacobs bisimulation.

For the Aczel/Mendler bisimulation we have to show, that there exists a function $r : R \longrightarrow (R \Rightarrow R) \Rightarrow R$ such that for instance

$$(T(R, \pi_2) \circ r)(m_1, m_1) \quad = \quad (T(\pi_2, M) \circ d)(m_1)$$

Using extensionality we derive, that for all functions $a : M \longrightarrow R$ it must hold that

$$\pi_2\bigl(r(m_1, m_1)(a \circ \pi_2)\bigr) \quad = \quad d(m_1)(\pi_2 \circ a)$$

This is impossible for f as defined at the beginning of this example, because we have $d(m_1)(\pi_2 \circ f) = m_5$ and further $r(m_1, m_1)(f \circ \pi_2) \in R$ and there is no pair in R that contains m_5.

One can argue, whether R as defined in this example should really be a bisimulation for c and d. The relevance of this example is, that it shows that the notion of Hermida/Jacobs bisimulation *does not* entail the notion of Aczel/Mendler bisimulation. ■

Higher-order polynomial functors can model arbitrary signatures over a type theory with products, coproducts and exponents. Thereby higher-order polynomial functors solve also the problem of binary methods in coalgebraic specification. The generality achieved is far beyond of what is necessary to model class interfaces of Eiffel (Meyer, 1992) or Java (Gosling et al., 1996). Only object-oriented extensions of functional programming languages (e.g., OCAML (Leroy et al., 2001)) require the full modelling power of higher-order polynomial functors.

The definitions for the terms invariant and bisimulation can be rather straightforward extended from the case of polynomial functors to higher-order polynomial functors. So higher-order polynomial functors and coalgebras for higher-order polynomial functors are a conservative extension of polynomial functors and coalgebras for polynomial functors, respectively. The price for the generality of higher-order polynomial functors is high: Almost none of well known properties from Section 2.6 for bisimulations and invariants for coalgebras of polynomial functors holds for the more general case. Counter examples can be constructed relatively easy.

3.4. Extended Polynomial Functors

The previous section showed that coalgebras for higher-order polynomial functors do not enjoy the same nice properties of coalgebras for polynomial functors. This section defines the class of *extended polynomial functors* that are a subclass of the higher-order polynomial functors. The idea of extended polynomial functors is based on the observation, that almost all of the counterexamples of the preceding section involve the functor T from Example 3.3.8. The subclass of extended polynomial functors restricts the exponents that can occur in the functors thereby excluding the functor T. Then I can prove many familiar results.

Definition 3.4.1 (Extended polynomial functors) Let $\mathbb{C}$ be a bicartesian closed category. A functor $G : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ is called *extended polynomial* if it is built according to the grammar

$$G(Y,X) \quad = \quad \begin{cases} X \\ A \\ G_1(Y,X) \times G_2(Y,X) \\ G_1(Y,X) + G_2(Y,X) \\ G_1(A',Y) \Rightarrow G_2(Y,X) \end{cases}$$

where A, A' are arbitrary objects of $\mathbb{C}$ and G_1 and G_2 are previously defined extended polynomial functors. The morphism part is defined in the obvious way:

$$G(g,f) \quad = \quad \begin{cases} f \\ \mathrm{id}_A \\ G_1(g,f) \times G_2(g,f) \\ G_1(g,f) + G_2(g,f) \\ G_1(\mathrm{id}_{A'}, g) \Rightarrow G_2(g,f) \end{cases} \qquad \text{if } G(Y,X) \;=\; \begin{cases} X \\ A \\ G_1(Y,X) \times G_2(Y,X) \\ G_1(Y,X) + G_2(Y,X) \\ G_1(A',Y) \Rightarrow G_2(Y,X) \end{cases}$$

The only (but crucial) difference compared to higher-order polynomial functors is the clause for the exponent. This accounts for the following fact: If G is an extended polynomial functor then, for each object A, there is a polynomial functor F such that $F(X) = G(A,X)$.

The functor T from Example 3.3.8 is not an extended polynomial functor. However, the functor NeighbourhoodPointIface (Example 3.2.4) fits into Definition 3.4.1. For object-oriented specification there is the following rule of thumb: A class signature gives rise to an extended polynomial functor, if all its methods have first order arguments (like the methods equal or move), or if for every functional argument it is the case, that Self does only occur in strictly positive positions in the type of the argument (like in the method move_with_neighbours).

Because every extended polynomial functor is also a higher-order polynomial functor, extended polynomial functors inherit the definitions of coalgebra (Definition 3.2.2), relation lifting (Definition 3.3.1), and bisimulation (Definition 3.3.3) from higher-order polynomial functors. Because of the restricted exponent these notions behave much more nicely.

The remainder of this section lists many results about coalgebras of extended polynomial functors. Basically all items from Fact 3.3.7 that have been answered negatively for higher-order polynomial functors can be answered positively for extended polynomial functors with only one exception: Invariants and bisimulations are not closed under union. I first concentrate on invariants and predicate lifting, next I turn to bisimulations and relation lifting. In Subsection 3.4.3 I consider bisimulations and coalgebra morphisms and in 3.4.4 I investigate the interplay of bisimulations and invariants. It follows a subsection on the relation of the Aczel/Mendler and the Hermida/Jacobs approach for extended polynomial functors. The last two subsections consider modifications in the definitions of invariant and bisimulation to obtain results about the union of invariants and the union of bisimulations, respectively.

3.4.1. Predicate Lifting and Invariants

Lemma 3.4.2 *Let G be an extended polynomial functor and assume two families of predicates $(P_i \subseteq X)_{i\in I}$ and $(Q_i \subseteq Y)_{i\in I}$. Then*

$$\textstyle\bigwedge_i \mathrm{Pred}(G)(Q_i, P_i) \quad \subseteq \quad \mathrm{Pred}(G)(\bigwedge_i Q_i, \bigwedge_i P_i)$$

Proof The proof method that I use for this result is quite important, it will be used for many other results in the following. First, I prove for polynomial functors F by induction on their structure the slightly stronger result

$$\textstyle\bigwedge_i \mathrm{Pred}(F)(P_i) \quad = \quad \mathrm{Pred}(F)(\bigwedge_i P_i) \qquad (*)$$

For the induction step I use Lemma 2.4.9 (1), for instance in case $F = A \Rightarrow F_1$:

$$\begin{array}{rcll}
\bigwedge_i \mathrm{Pred}(A \Rightarrow F_1)(P_i) & = & \bigwedge_i \big(\top_A \Rightarrow_{\mathrm{P}} \mathrm{Pred}(F_1)(P_i)\big) & \\
& = & \top_A \Rightarrow_{\mathrm{P}} \bigwedge_i \mathrm{Pred}(F_1)(P_i) & \text{by 2.4.9 (1)} \\
& = & \top_A \Rightarrow_{\mathrm{P}} \mathrm{Pred}(F_1)(\bigwedge_i P_i) & \text{by Ind. Hyp.} \\
& = & \mathrm{Pred}(A \Rightarrow F_1)(\bigwedge_i P_i) &
\end{array}$$

Now I prove the main result with induction on the structure of extended polynomial functors. In the induction steps I use Lemma 2.4.9 (1), 2.4.5 (1), and $(*)$. I demonstrate the case $G = G_1(A, Y) \Rightarrow G_2(Y, X)$. Assume, that the polynomial functor F equals $G_1(A, -)$.

$$\begin{array}{lll}
\multicolumn{3}{l}{\bigwedge_i \mathrm{Pred}(F \Rightarrow G_2)(Q_i, P_i)} \\
\quad = & \bigwedge_i \big(\mathrm{Pred}(F)(Q_i) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(G_2)(Q_i, P_i)\big) & \\
\quad \subseteq & \bigwedge_i \mathrm{Pred}(F)(Q_i) \Rightarrow_{\mathrm{P}} \bigwedge_i \mathrm{Pred}(G_2)(Q_i, P_i) & \text{by 2.4.9 (1)} \\
\quad = & \mathrm{Pred}(F)(\bigwedge_i Q_i) \Rightarrow_{\mathrm{P}} \bigwedge_i \mathrm{Pred}(G_2)(Q_i, P_i) & \text{by } (*) \\
\quad \subseteq & \mathrm{Pred}(F)(\bigwedge_i Q_i) \Rightarrow_{\mathrm{P}} & \text{by Ind. Hyp. and 2.4.5 (1)} \\
 & \qquad \mathrm{Pred}(G_2)(\bigwedge_i Q_i, \bigwedge_i P_i) & \\
\quad = & \mathrm{Pred}(F \Rightarrow G_2)(\bigwedge_i Q_i, \bigwedge_i P_i) & \square
\end{array}$$

Proposition 3.4.3 *For extended polynomial functors invariants are closed under arbitrary intersections.*

Proof Assume a collection $(P_i)_{i \in I}$ of invariants and also that $x \in \bigwedge_i P_i$. Then I have $c(x) \in \bigwedge_i \mathrm{Pred}(G)(P_i, P_i)$ and with the previous lemma $c(x) \in \mathrm{Pred}(G)(\bigwedge_i P_i, \bigwedge_i P_i)$ follows. Thus $\bigwedge_i P_i$ is indeed an invariant. □

The preceding proposition implies that all invariants for a given coalgebra form a complete lattice. So there exists a join operation on invariants that can be characterised as follows

$$P_1 \sqcup P_2 \qquad = \qquad \bigwedge \{Q \mid Q \text{ is an invariant and } P_1 \subseteq Q \text{ and } P_2 \subseteq Q\}$$

In general $P_1 \vee P_2$ is a proper subset of $P_1 \sqcup P_2$ for two invariants P_1 and P_2. Note that the greatest invariant contained in some predicate need not exist (because the join of all invariants contained in a predicate P might be greater than P). Greatest invariants are important in CCSL for the semantics of the infinitary modal operators always and eventually, see Section 4.5.2 (starting on page 173). CCSL uses therefore the notion of *strong invariant* that overcomes the just mentioned problems with Hermida/Jacobs invariants, see Subsection 3.4.6.

Let me continue to discuss the properties of Hermida/Jacobs invariants. For extended polynomial functors invariants give rise to subcoalgebras and vice versa. Technically I derive this result from Proposition 3.4.20, therefore the corresponding proposition is delayed until page 102.

Lemma 3.4.4

1. *Predicate lifting for polynomial functors F is cofibred:*

$$\mathrm{Pred}(F)(\textstyle\coprod_f P) \quad = \quad \textstyle\coprod_{F(f)} \mathrm{Pred}(F)(P)$$

for all predicates $P \subseteq X$ and functions $f : X \longrightarrow Y$.

2. *Predicate lifting for extended polynomial functors G is fibred:*

$$\mathrm{Pred}(G)(\textstyle\coprod_g Q, f^* P) \quad = \quad G(g,f)^* \,\mathrm{Pred}(G)(Q,P)$$

for all predicates $Q \subseteq U, P \subseteq Y$, and functions $g : U \longrightarrow V$, $f : X \longrightarrow Y$.

Proof First I prove (1) by induction on the structure of polynomial functors and use Lemma 2.4.17 (1) and (2). Let for instance $F = A \Rightarrow F_1$ be a polynomial functor and assume a suitable predicate P and a function f, then

$$\begin{array}{rcll} \mathrm{Pred}(F)(\coprod_f P) & = & \top_A \Rightarrow_{\mathrm{P}} \mathrm{Pred}(F_1)(\coprod_f P) & \\ & = & \top_A \Rightarrow_{\mathrm{P}} \coprod_{F_1(f)} \mathrm{Pred}(F_1)(P) & \text{by Ind. Hyp.} \\ & = & \coprod_{A \Rightarrow F_1(f)} \big(\top_A \Rightarrow_{\mathrm{P}} \mathrm{Pred}(F_1)(P)\big) & \text{by 2.4.17 (2)} \\ & = & \coprod_{F(f)} \mathrm{Pred}(F)(P) & \end{array}$$

Now I prove (2) by induction on the structure of extended polynomial functors. In the induction steps I use (1) and Lemma 2.4.15 (1). For instance if $G = F \Rightarrow G_1$ for a polynomial functor F, then

$$\begin{array}{rll} \multicolumn{3}{l}{\mathrm{Pred}(G)(\coprod_g Q, f^* P)} \\ = & \mathrm{Pred}(F)(\coprod_g Q) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(G_1)(\coprod_g Q, f^* P) & \\ = & \coprod_{F(g)} \mathrm{Pred}(F)(Q) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(G_1)(\coprod_g Q, f^* P) & \text{by (1)} \\ = & \coprod_{F(g)} \mathrm{Pred}(F)(Q) \Rightarrow_{\mathrm{P}} G_1(g,f)^* \,\mathrm{Pred}(G_1)(Q,P) & \text{by Ind. Hyp.} \\ = & (F(g) \Rightarrow G_1(g,f))^* \big(\mathrm{Pred}(F)(Q) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(G_1)(Q,P)\big) & \\ & & \text{by 2.4.15 (1)} \\ = & G(g,f)^* \,\mathrm{Pred}(G)(Q,P) & \Box \end{array}$$

3.4.2. Relation Lifting and Bisimulations

I turn now to properties of relation lifting and bisimulations for extended polynomial functors.

Lemma 3.4.5 *Let G be an extended polynomial functor.*

1. *Let I be an arbitrary nonempty index set and assume two indexed collections $(R_i \subseteq X \times Y)_{i \in I}$ and $(S_i \subseteq U \times V)_{i \in I}$ of relations. Then we have:*

$$\textstyle\bigwedge_i \mathrm{Rel}(G)(S_i, R_i) \quad \subseteq \quad \mathrm{Rel}(G)(\bigwedge_i S_i, \bigwedge_i R_i)$$

2. *Assume now relations $R_1 \subseteq X \times Y$, $R_2 \subseteq Y \times Z$, $S_1 \subseteq U \times V$ and $S_2 \subseteq V \times W$, then:*

$$\mathrm{Rel}(G)(S_1, R_1) \;\circ\; \mathrm{Rel}(G)(S_2, R_2) \quad \subseteq \quad \mathrm{Rel}(G)(S_1 \circ S_2, R_1 \circ R_2)$$

Proof I use the same proof method as before. For (1) I first prove with the help of Lemma 2.4.9 (2) for polynomial functors F

$$\textstyle\bigwedge_i \mathrm{Rel}(F)(R_i) \quad = \quad \mathrm{Rel}(F)(\bigwedge_i R_i) \qquad (*)$$

by induction on their structure. For instance if $F(X) = A \Rightarrow F_1(X)$ then

$$\begin{aligned}
\textstyle\bigwedge_i \mathrm{Rel}(A \Rightarrow F_1)(R_i) &= \textstyle\bigwedge_i \big(\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \mathrm{Rel}(F_1)(R_i)\big) \\
&= \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \textstyle\bigwedge_i \mathrm{Rel}(F_1)(R_i) && \text{by 2.4.9 (2)} \\
&= \mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \mathrm{Rel}(F_1)(\textstyle\bigwedge_i R_i) && \text{by Ind. Hyp.} \\
&= \mathrm{Rel}(A \Rightarrow F_1)(\textstyle\bigwedge_i R_i)
\end{aligned}$$

For the result 3.4.5 (1) I do induction on the structure of G. To apply the induction hypothesis I additionally need Lemma 2.4.5 (2). Again I show the most difficult induction step for $G = F \Rightarrow G_1$ with a polynomial functor F in detail.

$$\begin{aligned}
&\textstyle\bigwedge_i \mathrm{Rel}(F \Rightarrow G_1)(S_i, R_i) \\
&\quad= \textstyle\bigwedge_i \big(\mathrm{Rel}(F)(S_i) \Rightarrow_{\mathrm{R}} \mathrm{Rel}(G_1)(S_i, R_i)\big) \\
&\quad\subseteq \textstyle\bigwedge_i \mathrm{Rel}(F)(S_i) \Rightarrow_{\mathrm{R}} \bigwedge_i \mathrm{Rel}(G_1)(S_i, R_i) && \text{by 2.4.9 (2)} \\
&\quad= \mathrm{Rel}(F)(\textstyle\bigwedge_i S_i) \Rightarrow_{\mathrm{R}} \bigwedge_i \mathrm{Rel}(G_1)(S_i, R_i) && \text{by } (*) \\
&\quad\subseteq \mathrm{Rel}(F)(\textstyle\bigwedge_i S_i) \Rightarrow_{\mathrm{R}} \mathrm{Rel}(G_1)(\bigwedge_i S_i, \bigwedge_i R_i) && \text{by Ind. Hyp. and 2.4.5 (2)} \\
&\quad= \mathrm{Rel}(F \Rightarrow G_1)(\textstyle\bigwedge_i S_i, \bigwedge_i R_i)
\end{aligned}$$

For (2) I use the same proof method: Using Lemma 2.4.12 I first establish an equality for polynomial functors. I can then prove the main result by induction on the structure of extended polynomial functors. □

Proposition 3.4.6 *Bisimulations for extended polynomial functors are closed under arbitrary nonempty intersection and composition.*

Note, that bisimulations for extended polynomial functors are not closed under union in general. Different to invariants, bisimulations do not form a complete lattice. There is no greatest bisimulation in general, see Example 3.3.9.

Proof Both proofs are very similar. I only do composition here. I have to show, that for an arbitrary extended polynomial functor G, three G–coalgebras $c : X \longrightarrow G(X, X)$, $d : Y \longrightarrow G(Y, Y)$, $e : Z \longrightarrow G(Z, Z)$, a G–bisimulation $R \subseteq X \times Y$ for c and d and a G–bisimulation $S \subseteq Y \times Z$ for d and e also their composition $R \circ S$ is a G–bisimulation for c and e. With the definition of bisimulation it remains to show, that for all $x \in X$, $y \in Y$, $z \in Z$ such that $R(x, y)$, $S(y, z)$, $\mathrm{Rel}(G)(R, R)(c(x), d(y))$ and $\mathrm{Rel}(G)(S, S)(d(y), e(z))$ it is also the case, that $\mathrm{Rel}(G)(R \circ S, R \circ S)(c(x), e(z))$. This follows directly from the previous Lemma. □

3.4.3. Bisimulations and Coalgebra Morphisms

The next lemma leads to Proposition 3.4.8 about bisimulations and morphisms.

Lemma 3.4.7

1. *The relation lifting for polynomial functors is cofibred: for a polynomial functor F, a relation $R \subseteq X \times Y$ and two functions $f : X \longrightarrow X'$, $g : Y \longrightarrow Y'$ it holds that*

$$\mathrm{Rel}(F)(\textstyle\coprod_{f\times g} R) \quad = \quad \textstyle\coprod_{F(f)\times F(g)} \mathrm{Rel}(F)(R)$$

2. *The relation lifting for extended polynomial functors is fibred: for an extended polynomial functor G, two relations $S \subseteq U \times V$, $R \subseteq X \times Y$ and four functions $u : U \longrightarrow U'$, $v : V \longrightarrow V'$, $f : X' \longrightarrow X$, $g : Y' \longrightarrow Y$ it holds that*

$$\mathrm{Rel}(G)(\textstyle\coprod_{u\times v} S,\, (f\times g)^* R) \quad = \quad \big(G(u,f)\times G(v,g)\big)^* \,\mathrm{Rel}(G)(S,R)$$

Proof The proof is completely analogous to the proof of Lemma 3.4.4. □

Proposition 3.4.8 *Let G be an extended polynomial functor and $c : X \longrightarrow G(X,X)$ and $d : Y \longrightarrow G(Y,Y)$ be two G–coalgebras. A function $f : X \longrightarrow Y$ is a morphism between c and d if and only if the graph of f given by $\mathsf{graph}(f) = \coprod_{\mathrm{id}_X\times f} \mathrm{Eq}(X)$ is a bisimulation for c and d.*

Proof I have to show that for a function $f : X \longrightarrow Y$ an arbitrary pair $(c\,x, d(f\,x))$ is in $\mathrm{Rel}(G)(\mathsf{graph}(f), \mathsf{graph}(f))$ if and only if $G(X,f)(c\,x) = G(f,Y)(d(f\,x))$. The latter is equivalent with $(c\,x, d(f\,x)) \in (G(X,f)\times G(f,Y))^*\,\mathrm{Eq}(G(X,Y))$.

First recall the equivalence stated in Equation 2.7 (on page 64):

$$\textstyle\coprod_{\mathrm{id}_X\times f} \mathrm{Eq}(X) \quad = \quad (f\times \mathrm{id}_Y)^*\,\mathrm{Eq}(Y) \qquad (*)$$

Using this intermediate result I compute

$$\begin{array}{lll}
\multicolumn{2}{l}{\mathrm{Rel}(G)(\mathsf{graph}(f), \mathsf{graph}(f))} & \\
= & \mathrm{Rel}(G)\big(\textstyle\coprod_{\mathrm{id}_X\times f} \mathrm{Eq}(X), (f\times \mathrm{id}_Y)^*\,\mathrm{Eq}(Y)\big) & \text{by } (*) \\
= & (G(\mathrm{id}_X, f)\times G(f,\mathrm{id}_Y))^*\,\mathrm{Rel}(G)(\mathrm{Eq}(X), \mathrm{Eq}(Y)) & \text{by 3.4.7} \\
= & (G(\mathrm{id}_X, f)\times G(f,\mathrm{id}_Y))^*\,\mathrm{Eq}(G(X,Y)) & \text{by 3.3.2 (3)}
\end{array}$$ □

The next result is an immediate consequence of the Proposition 3.4.8 and 3.4.6.

Proposition 3.4.9 *The kernel of an arbitrary morphism between G–coalgebras is a bisimulation for every extended polynomial functor G.* □

3.4.4. Bisimulations and Invariants

In this subsection I consider the constructions on bisimulations and invariants from Subsection 2.6.5. The first lemma is towards the result that $\coprod_{\pi_1} R$ is an invariant if R is a bisimulation.

Lemma 3.4.10 *Let G be an extended polynomial functor and R and S be arbitrary relations. Then*

$$\coprod_{\pi_1} \mathrm{Rel}(G)(S, R) \quad\subseteq\quad \mathrm{Pred}(G)(\coprod_{\pi_1} S,\ \coprod_{\pi_1} R)$$

Proof By induction on the structure of G using Lemma 2.6.14 in the induction step for the exponent. The induction steps of this proof have been formalised in PVS. □

Proposition 3.4.11 *Assume two coalgebras $c : X \longrightarrow G(X, X)$ and $d : Y \longrightarrow G(Y, Y)$ for an extended polynomial functor G. If $R \subseteq X \times Y$ is a bisimulation for c and d then the predicate $\coprod_{\pi_1} R \subseteq X$ is an invariant for c.*

Proof Along the lines of Proposition 2.6.15. □

Example 3.4.12 In this example I show that the previous result does not hold in general for higher-order polynomial functors. I construct a bisimulation for the higher-order polynomial functor T such that its first projection is not an invariant. The functor T is defined in Example 3.3.8 on page 88.

Take $X \overset{\mathrm{def}}{=} \{x_1, x_2, x_3, x_4\}$ and $Y \overset{\mathrm{def}}{=} \{y_1, y_2\}$. Define the function f and the T–coalgebras c and d as follows.

$$\begin{aligned}
f &: X \longrightarrow X \\
&= \lambda x : X \,.\, \mathsf{if}\ x = x_2\ \mathsf{then}\ x_3\ \mathsf{else}\ x_1\ \mathsf{endif} \\
c &: X \longrightarrow (X \Rightarrow X) \Rightarrow X \\
&= \lambda x : X \,.\, \lambda h : X \longrightarrow X \,.\, \mathsf{if}\ h = f\ \mathsf{then}\ x_4\ \mathsf{else}\ x_1\ \mathsf{endif} \\
d &: Y \longrightarrow (Y \Rightarrow Y) \Rightarrow Y \\
&= \lambda y : Y \,.\, \lambda g : Y \longrightarrow Y \,.\, y_1
\end{aligned}$$

Similar to Example 3.3.12 the relation $R \overset{\mathrm{def}}{=} \{(x_1, y_1),\ (x_2, y_1),\ (x_3, y_2)\}$ is a bisimulation for c and d because there is no function that is (R, R)–related to f. Its first projection $\coprod_{\pi_1} R = \{x_1, x_2, x_3\}$ is not an invariant for c. ■

The next lemma is towards the result that the intersection of an invariant P with a bisimulation R, precisely $R \wedge \pi_1^* P = \{(x, y) \mid R(x, y) \wedge P(x)\}$, is a bisimulation again.

Lemma 3.4.13 *Let G be an extended polynomial functor, S and R be arbitrary relations, and P and Q arbitrary predicates. Then*

$$\mathrm{Rel}(G)(S,R) \;\wedge\; \pi_1^*\left(\mathrm{Pred}(G)(Q,P)\right) \quad\subseteq\quad \mathrm{Rel}(G)(S \wedge \pi_1^*(Q),\; R \wedge \pi_1^*(P))$$

Proof By induction on the structure of G using Lemma 2.6.16 in the induction step for the exponent. The induction steps of this proof have been formalised in PVS. □

Proposition 3.4.14 *Let $c : X \longrightarrow G(X,X)$ and $d : Y \longrightarrow G(Y,Y)$ be two coalgebras for an extended polynomial functor G. Assume that $R \subseteq X \times Y$ is a bisimulation for c and d and that $P \subseteq X$ is an invariant for c. Then the relation $R \wedge \pi_1^* P$ is a bisimulation for c and d.*

Proof Along the lines of 2.6.17. □

Example 3.4.15 In this example I construct a bisimulation and an invariant for the functor T from Example 3.3.8 such that their intersection is not a bisimulation. Let $X \overset{\mathrm{def}}{=} \{x_1, x_2\}$ and $Y \overset{\mathrm{def}}{=} \{y_1, y_2, y_3\}$. Define the function g and the T–coalgebras c and d as follows

$$\begin{array}{rcl} g & : & Y \longrightarrow Y \\ & = & \lambda y : Y \,.\, \mathsf{if}\ y = y_2\ \mathsf{then}\ y_3\ \mathsf{else}\ y_1\ \mathsf{endif} \\ c & : & X \longrightarrow (X \Rightarrow X) \Rightarrow X \\ & = & \lambda x : X \,.\, \lambda h : X \longrightarrow X \,.\, x_1 \\ d & : & Y \longrightarrow (Y \Rightarrow Y) \Rightarrow Y \\ & = & \lambda y : Y \,.\, \lambda k : Y \longrightarrow Y \,.\, \mathsf{if}\ k = g\ \mathsf{then}\ y_3\ \mathsf{else}\ y_1\ \mathsf{endif} \end{array}$$

The relation $R \overset{\mathrm{def}}{=} \{(x_1, y_1),\ (x_2, y_2)\}$ is a bisimulation for c and d because there is no function that is (R,R)–related to g. The predicate $P \overset{\mathrm{def}}{=} \{x_1\}$ is clearly an invariant for c. Set now $S \overset{\mathrm{def}}{=} R \cap \pi_1^* P = \{(x_1, y_1)\}$. There is function that is (S,S)–related to g, namely id_X, therefore S is not a bisimulation for c and d. ■

With the previous two propositions and with Proposition 3.4.8 it is possible to derive more results in the same way as for coalgebras of polynomial functors.

Proposition 3.4.16 *Let G be an extended polynomial functor and let $f : c \longrightarrow d$ be a morphism between two coalgebras $c : X \longrightarrow G(X,X)$ and $d : Y \longrightarrow G(Y,Y)$.*

1. *If $P \subseteq X$ is an invariant for c then $\coprod_f P$ is an invariant for d.*
2. *If $Q \subseteq Y$ is an invariant for d then $f^* Q$ is an invariant for c.*

Proof Almost identical to the proof of 2.6.18. □

Example 3.4.17 This example shows a coalgebra morphism for the higher-order polynomial functor T (see Example 3.3.8 on page 88) such that the image of this morphism is not an invariant. Take $X \overset{\text{def}}{=} \{x_1, x_2\}$ and $Y \overset{\text{def}}{=} \{y_1, y_2\}$ and define the functions f and g and the T–coalgebras c and d as follows.

$$\begin{array}{lcl} f & : & X \longrightarrow Y \\ & = & \lambda x : X \,.\, y_1 \\[1ex] g & : & Y \longrightarrow Y \\ & = & \lambda y : Y \,.\, y \\[1ex] c & : & X \longrightarrow (X \Rightarrow X) \Rightarrow X \\ & = & \lambda x : X \,.\, \lambda h : X \longrightarrow X \,.\, x \\[1ex] d & : & Y \longrightarrow (Y \Rightarrow Y) \Rightarrow Y \\ & = & \lambda y : Y \,.\, \lambda k : Y \longrightarrow Y \,.\, \mathsf{if}\ k = g\ \mathsf{then}\ y_2\ \mathsf{else}\ y_1\ \mathsf{endif} \end{array}$$

The function f is a morphism $c \longrightarrow d$. The truth predicate $\top_X$ is clearly an invariant for c. However, the predicate $\coprod_f \top_X = \{y_1\}$ is not an invariant for d. ■

The next proposition is inspired by (Rutten, 2000).

Proposition 3.4.18 *Let $c : X \longrightarrow G(X, X)$ be a coalgebra of an extended polynomial functor G. A predicate $P \subseteq X$ is an invariant for c if and only if the diagonal on P, the relation $\coprod_\delta P$, is a bisimulation for c.*

Proof The proof is identical to 2.6.19. The *if* part follows from Proposition 3.4.11. The *only if* part from Propositions 3.4.14 and 3.3.6 (2). □

Example 3.4.19 This example shows a coalgebra c for the higher-order polynomial functor T (see example 3.3.8 on page 88) and an invariant P for c such that $\coprod_\delta P$ is not a bisimulation for c. Take $X \overset{\text{def}}{=} \{x_1, x_2, x_3\}$ and define the functions f, g, and c as follows.

$$\begin{array}{lcl} f & : & X \longrightarrow X \\ & = & \lambda x : X \,.\, \mathsf{if}\ x = x_3\ \mathsf{then}\ x_1\ \mathsf{else}\ x \\[1ex] g & : & X \longrightarrow X \\ & = & \lambda x : X \,.\, \mathsf{if}\ x = x_3\ \mathsf{then}\ x_2\ \mathsf{else}\ x \\[1ex] c & : & X \longrightarrow (X \Rightarrow X) \Rightarrow X \\ & = & \lambda x : X \,.\, \lambda h : X \longrightarrow X \,.\, \mathsf{if}\ h = f\ \mathsf{then}\ x_2\ \mathsf{else}\ x_1 \end{array}$$

The predicate $P \overset{\text{def}}{=} \{x_1, x_2\}$ is an invariant for c. Let $R \overset{\text{def}}{=} \coprod_\delta P = \{(x_1, x_1), (x_2, x_2)\}$. The functions f and g are (R, R)–related but $c\,x_1\,f = x_2 \neq x_1 = c\,x_1\,g$. This is why R is not a bisimulation for c. ■

3.4.5. The Aczel/Mendler Approach Revisited

In this subsection I consider the relation of the Hermida/Jacobs and the Aczel/Mendler approach for extended polynomial functors.

Proposition 3.4.20 *For extended polynomial functors the notions of Aczel/Mendler bisimulation and Hermida/Jacobs bisimulation coincide.*

Proof Assume, that G is an extended polynomial functor, $c : X \longrightarrow G(X, X)$ and $d : Y \longrightarrow G(Y, Y)$ are two G–coalgebras, and $R \subseteq X \times Y$ is a relation. Consider the following properties

$$\begin{aligned}&\forall x \in G(X,X),\ y \in G(Y,Y),\ r \in G(R,R)\,.\\ &\qquad G(R,\pi_1)(r) = G(\pi_1, X)(x) \ \text{ and } \ G(R,\pi_2)(r) = G(\pi_2, Y)(y) \\ &\qquad\qquad \text{implies } \ \mathrm{Rel}(G)(R,R)(x,y)\end{aligned} \tag{$*$}$$

$$\begin{aligned}&\forall x \in G(X,X),\ y \in G(Y,Y),\\ &\qquad \mathrm{Rel}(G)(R,R)(x,y) \ \text{ implies } \ \exists r \in G(R,R)\,.\\ &\qquad\qquad G(R,\pi_1)(r) = G(\pi_1, X)(x) \ \text{ and } \ G(R,\pi_2)(r) = G(\pi_2, Y)(y)\end{aligned} \tag{$\dagger$}$$

If R is an Aczel/Mendler bisimulation, then $(*)$ implies that R is also a Hermida/Jacobs bisimulation. If, for the other direction, R is a Hermida/Jacobs bisimulation, then $(\dagger)$ implies that for each $(a, b) \in R$ there exists a suitable element in $G(R, R)$. It is therefore possible to construct the needed function $R \longrightarrow G(R, R)$ for the Aczel/Mendler bisimulation using the Axiom of Choice.

It remains to show, that $(*)$ and $(\dagger)$ hold for all extended polynomial functors G and all relations R. This can be done by induction on the structure of G. The base cases and the induction steps for $\times_{\mathrm{R}}$ and $+_{\mathrm{R}}$ are easy computations. For the exponent, the Axiom of Choice is needed again. For this proof, the predicates $(*)$ and $(\dagger)$ and the induction steps have been formalised in PVS. □

Proposition 3.4.21 *For extended polynomial functors the notions of Hermida/Jacobs invariant and Aczel/Mendler invariant coincide.*

Proof Let G be an extended polynomial functor, $c : X \longrightarrow G(X, X)$ be a G–coalgebra and $P \subseteq X$ be a predicate. Set $R \overset{\text{def}}{=} \mathrm{Eq}(X) \wedge \pi_1^* P$, then by Frobenius we also have

$\coprod_{\pi_1} R = P$. The projection π_1 restricts to an isomorphism $\pi_1 : R \longrightarrow P$. Consider now the following diagram.

$$\begin{tikzcd}
X \arrow[r, "c"] & G(X,X) \arrow[rd, "G(\iota,X)"'] \arrow[rrrdd, rounded corners, to path={-| (\tikztotarget)}, "G(\pi_1,X)"] & & & \\
& (*) & G(P,X) \arrow[rrd, "\cong", "G(\pi_1,X)"'] & \circlearrowright & \\
P \arrow[uu, "\iota"] \arrow[r, "p"] & G(P,P) \arrow[ru, "G(P,\iota)"] \arrow[rd, "\cong", "G(\pi_1,P)"'] & \circlearrowright & & G(R,X) \\
& & G(R,P) \arrow[rru, "G(R,\iota)"] & \circlearrowright & \\
R \arrow[uu, "\pi_1", "\cong"'] \arrow[r, "r"] & G(R,R) \arrow[ru, "G(R,\pi_1)", "\cong"'] \arrow[rrruu, rounded corners, to path={-| (\tikztotarget)}, "G(R,\pi_1)"'] & & &
\end{tikzcd}$$

The outer pentagon commutes, if R is an Aczel/Mendler bisimulation with witness r. The upper left pentagon $(*)$ commutes if P is an Aczel/Mendler invariant with witness p. The arrows marked with $\cong$ are isomorphisms. The parts of the diagram marked with $\circlearrowright$ commute always because G is a functor.

Let us now tackle the proposition: It remains to show that there is an equivalence: P is a Hermida/Jacobs invariant if and only if there exists $p : P \longrightarrow G(P,P)$ such that $(*)$ commutes. Consider first the *only if* case:

P is a Hermida/Jacobs invariant

$\Longrightarrow$ by 3.4.14 $R = \mathrm{Eq}(X) \wedge \pi_1^* P$ is a Hermida/Jacobs bisimulation

$\Longrightarrow$ by 3.4.20 R is an Aczel/Mendler bisimulation (i.e., there exists $r : R \longrightarrow G(R,R)$ making the outer pentagon commute)

$\Longrightarrow$ by diagram chasing $(*)$ commutes if $p = G(\pi_1,P)^{-1} \circ G(R,\pi_1) \circ r \circ {\pi_1}^{-1}$

$\Longrightarrow$ P is an Aczel/Mendler invariant

The line of reasoning works also in the other direction: If we are given $p : P \longrightarrow G(P,P)$ such that $(*)$ commutes we set $r = G(R,\pi_1)^{-1} \circ G(\pi_1,P) \circ p \circ \pi_1$ and get that the outer pentagon commutes. With 3.4.20 and 3.4.11 we get finally that P is a Hermida/Jacobs invariant. □

3.4.6. Strong Invariants

In this subsection I present the notion of invariant that is used in the context of CCSL.

Definition 3.4.22 (Strong Invariant) Let H be a higher-order polynomial functor and $c : X \longrightarrow H(X,X)$ be an H–coalgebra. A predicate $P \subseteq X$ is a *strong H–invariant* for c if for all $x \in X$

$$P(x) \quad \text{implies} \quad \mathrm{Pred}(H)(\top_X, P)(c(x))$$

The only difference to Definition 3.3.3 is that the contravariant argument of $\mathrm{Pred}(H)$ is now instantiated with $\top_X$ instead of P. For polynomial functors there is no difference between strong invariants and Hermida/Jacobs invariants. For higher-order polynomial functors the difference can be explained intuitively as follows: Let $\mathsf{bm} : \mathsf{Self} \times \mathsf{Self} \longrightarrow \mathsf{Self}$ be a binary method, P be a predicate on Self, and $x \in P$. A strong invariant requires that $\mathsf{bm}(x, y) \in P$ for all possible y. The definition of Hermida/Jacobs invariant requires $\mathsf{bm}(x, y) \in P$ only if $y \in P$.

Proposition 3.4.23

Every strong invariant is a Hermida/Jacobs invariant.

Proof Immediate from Lemma 3.3.2 (1). □

The interplay between strong invariants and bisimulations is a bit intricate. Proposition 3.4.14 ($R \wedge \pi_1^* P$ is a bisimulation) holds if one substitutes strong invariant for invariant. Obviously, Proposition 3.4.11 ($\coprod_{\pi_1} R$ is an invariant) does not hold for strong invariants. For Proposition 3.4.18 (P is an invariant if and only if $\coprod_\delta P$ is a bisimulation) the only–if part holds for strong invariants (by the preceding proposition). Obviously, the if part is not true. Finally, considering Proposition 3.4.16 (1) one finds that the predicate $\coprod_f P$ is an invariant for every strong invariant P, but $\coprod_f P$ might not be a strong invariant. Item 2 of this proposition is clarified for strong invariants with the following result.

Proposition 3.4.24 *Let G be an extended polynomial functor and let $f : c \longrightarrow d$ be a morphism between two coalgebras $c : X \longrightarrow G(X, X)$ and $d : Y \longrightarrow G(Y, Y)$. If $Q \subseteq Y$ is a strong invariant for d then $f^* Q$ is a strong invariant for c.*

Proof If Q is a strong invariant for d, then

$$Q \quad \subseteq \quad d^* \,\mathrm{Pred}(G)(\top_Y, Q) \qquad (*)$$

It remains to show that $f^* Q \subseteq c^* \,\mathrm{Pred}(G)(\top_X, f^* Q)$. The derivation is as follows:

$$\begin{array}{rcll}
f^* Q & \subseteq & f^* d^* \,\mathrm{Pred}(G)(\top_Y, Q) & \text{by } (*) \text{ and monotonicity of } f^* \\
& \subseteq & f^* d^* \,\mathrm{Pred}(G)(\coprod_f \top_X, Q) & \coprod_f \top_X \subseteq \top_Y \\
& = & f^* d^* \,G(f, Y)^* \,\mathrm{Pred}(G)(\top_X, Q) & \text{by 3.4.4 (2)} \\
& = & c^* \,G(X, f)^* \,\mathrm{Pred}(G)(\top_X, Q) & G(f, Y) \circ d \circ f \;=\; G(X, f) \circ c \\
& = & c^* \,\mathrm{Pred}(G)(\top_X, f^* Q) & \text{by 3.4.4 (2)} \quad \square
\end{array}$$

In the following I show that strong invariants are closed under arbitrary union and intersection for higher-order polynomial functors and that they allow the construction of the greatest strong invariant contained in some predicate (i.e., the strong invariants contained in some predicate form a complete lattice). Let $c : X \longrightarrow H(X, X)$ be a coalgebra for a higher-order polynomial functor and let $P \subseteq X$ be a predicate on X. The greatest strong invariant[6] contained in P is denoted with $\underline{P}$. Obviously, if the greatest invariant exists then it is unique. Consider now the following function $\Phi_P : \mathbf{Pred}_X \longrightarrow \mathbf{Pred}_X$ between predicates over X (for a fixed coalgebra c):

$$\Phi_P(Q \subseteq X) \quad \overset{\text{def}}{=} \quad P \wedge c^* \left(\mathrm{Pred}(H)(\top_X, Q)\right)$$

Let Q be a fixed point of Φ_P, that is $\Phi_P(Q) = Q$. Then we have $Q \subseteq c^* \left(\mathrm{Pred}(H)(\top_X, Q)\right)$ which is the defining condition for strong invariants in different notation. So any fixed point of Φ_P is a strong invariant.

Proposition 3.4.25 *Let H be a higher-order polynomial functor and $c : X \longrightarrow H(X, X)$ be a H–coalgebra. For every predicate $P \subseteq X$ the greatest strong invariant contained in P exists, it is given as the greatest fixed point of Φ_P.*

Proof For the application of the Knaster/Tarski fixed point theorem (Tarski, 1955) I have to show that Φ_P is monotone, that is that $Q_1 \subseteq Q_2$ implies $\Phi_P(Q_1) \subseteq \Phi_P(Q_2)$. From Lemma 3.3.2 (1) I get $\mathrm{Pred}(H)(\top_X, Q_1) \subseteq \mathrm{Pred}(H)(\top_X, Q_2)$, and monotonicity of Φ_P follows from the monotonicity of $\wedge$. Every strong invariant is a prefixed point of Φ_P, so the result follows from Knaster/Tarski. □

Note that I just proved that Φ_P is an endofunctor on $\mathbf{Pred}_X$. The existence of greatest strong invariants ensures the semantics for the infinitary model operator always and eventually in CCSL, see Subsection 4.5.2 on page 173.

Lemma 3.4.26 *Consider a higher-order polynomial functor H and an arbitrary family of predicates $(P_i \subseteq X)_{i \in I}$, then*

$$\begin{array}{lcl} \bigwedge_i \mathrm{Pred}(H)(\top_Y, P_i) & \subseteq & \mathrm{Pred}(H)(\top_Y, \bigwedge_i P_i) \\ \bigvee_i \mathrm{Pred}(H)(\top_Y, P_i) & \subseteq & \mathrm{Pred}(H)(\top_Y, \bigvee_i P_i) \end{array}$$

Proof First prove

$$\mathrm{Pred}(H)(Q, \top) \quad = \quad \top \qquad (*)$$

by induction on H. The two main results follows now also by induction on H. The cases of product and coproduct follow directly from the Lemmas 2.4.10 (1), 2.4.9 (1), and 2.4.5 (1). For the case $H = H_1 \Rightarrow H_2$ note that $(*)$ implies $\mathrm{Pred}(H_1 \Rightarrow H_2)(\top_Y, P_i) = \mathrm{Pred}(H_1)(P_i, \top_Y) \Rightarrow_{\mathrm{P}} \mathrm{Pred}(H_2)(\top_Y, P_i) = \top_{H_1(X,Y)} \Rightarrow_{\mathrm{P}} \mathrm{Pred}(\top_Y, H_2)(P_i)$. □

[6] Compare Subsection 2.6.6 on greatest invariants of coalgebras for polynomial functors.

Proposition 3.4.27 *For coalgebras of higher-order polynomial functors strong invariants are closed under arbitrary intersection and union.*

Proof Apply the preceding lemma. □

Just for completeness, let me state the following characterisation for greatest strong invariants (which follows now immediately):

$$\underline{Q} \qquad = \qquad \bigvee \{P \mid P \subseteq Q \;\text{ and } P \text{ is a strong invariant}\}$$

The advantage of strong invariants is that the greatest invariant contained in some predicate does always exist. This allows me to give semantics to the infinitary modal operators always and $\mathsf{eventually}$ in Subsection 4.5.2. The disadvantage is that strong invariants do not fit well together with bisimulations.

3.4.7. Partially Reflexive Bisimulations

Let $c : X \longrightarrow G(X, X)$ be a coalgebra for an extended polynomial functor G and consider bisimulations for c (i.e., relations $R \subseteq X \times X$ that are bisimulations for c and c). One expects that a notion that captures behavioural indistinguishability is an equivalence: if $x \in X$ and $y \in X$ show the same behaviour and y and $z \in X$ show the same behaviour, then also x and z show the same behaviour. However, the usual approach to define the notion of bisimulation for coalgebras does not require bisimulations to be transitive, symmetric, or reflexive. Instead one shows that for every bisimulation R (for a polynomial functor) the least equivalence relation $\overline{R}$ containing R is a bisimulation again.

This approach fails for extended polynomial functors, see Example 3.3.9. In this subsection and in the next section on extended cartesian functors I consider bisimulations with additional properties (for instance bisimulations that are reflexive) that behave more nicely with respect to union. First I have to fix some notation.

Definition 3.4.28 Let $R \subseteq X \times X$ be a relation.

- The *domain* of R is the predicate $\mathsf{dom}(R)$, defined as

$$\mathsf{dom}(R) \quad = \quad \textstyle\coprod_{\pi_1} R \vee \coprod_{\pi_2} R \quad = \quad \{x \in X \mid \exists y \in X \,.\, x\,R\,y \vee y\,R\,x\}$$

- The relation R is *partially reflexive* if it is reflexive on its domain, that is, if for all $x, y \in X$

$$x\,R\,y \qquad \text{implies} \qquad x\,R\,x \quad \text{and} \quad y\,R\,y$$

The following result is immediate.

Lemma 3.4.29 *Let $R, S \subseteq X \times X$ be partially reflexive relations on the same domain, then $R \vee S \subseteq R \circ S$* □

This gives immediately the following proposition.

Proposition 3.4.30 *Let $c : X \longrightarrow G(X,X)$ be a coalgebra for an extended polynomial functor G and let $R, S \subseteq X \times X$ be bisimulations for c. If R and S are partially reflexive on the same domain, then there exists a bisimulation containing $R \vee S$. In particular, there is a bisimulation containing $R \vee S$ if both R and S are reflexive.*

Proof Combining Proposition 3.4.6 with the preceding lemma yields $R \circ S$ as the required bisimulation. □

One should expect that any decent notion of behavioural indistinguishability is reflexive, so restricting the attention to reflexive (or even only to partially reflexive) bisimulations seems *very* reasonable. Note, that the preceding theorem does *not* yield the existence of a greatest bisimulation for c. For the greatest bisimulation one needs an upper bound for arbitrary collections of bisimulations, whereas Proposition 3.4.30 only yields an upper bound for finite collections. There are indeed examples with infinitely ascending chains of bisimulations, see Example 3.5.10.

Let me now turn to bisimulations between two coalgebras $c : X \longrightarrow G(X,X)$ and $d : Y \longrightarrow G(Y,Y)$. Consider two relations $R, S \subseteq X \times Y$. A relation $B \subseteq X \times Y$ is called a *base* for R and S if all the following points hold:

$$\begin{array}{llcll} (1) & B \subseteq R & \qquad & (3) & \coprod_{\pi_2} R \;=\; \coprod_{\pi_2} B \\ (2) & B \subseteq S & & (4) & \coprod_{\pi_1} S \;=\; \coprod_{\pi_1} B \end{array}$$

The existence of a base is a particular generalisation of the notion of partial reflexiveness: If R and S are partially reflexive on the same domain then the equality relation restricted to the common domain is a base for R and S.

Proposition 3.4.31 *Let $c : X \longrightarrow G(X,X)$ and $d : Y \longrightarrow G(Y,Y)$ be two coalgebras for an extended polynomial functor G and let R and $S \subseteq X \times Y$ be two bisimulations for c and d with a base B. If B is a bisimulation then there exists a bisimulation containing $R \vee S$.*

Proof Being a base for R and S is just the technical condition that allows one to derive $R \vee S \;\subseteq\; R \circ B^{\mathrm{op}} \circ S$. Now one can apply Proposition 3.4.6. □

This last proposition is on the edge of my knowledge at the time of writing. It remains an open question for future work if Proposition 3.4.31 is useful in applications. It is mainly included here to demonstrate that the essential property that allows the proof of Proposition 3.4.30 can be generalised in a straightforward way.

3.5. Extended Cartesian Functors

Extended cartesian functors are the least generalisation of polynomial functors that I consider in the present thesis. They deserve their own section because they allow me to adopt the result from (Poll and Zwanenburg, 2001) that (partial) bisimulation equivalences form a complete lattice. Let me first give some definitions to make the discussion easier.

Definition 3.5.1 (Extended Cartesian Functors) Assume a bicartesian closed category $\mathbb{C}$.

- A *cartesian functor* is a polynomial functor without exponent. More precisely, a functor $K : \mathbb{C} \longrightarrow \mathbb{C}$ is a cartesian functor if it is defined as one of the cases

$$K(X) \quad = \quad \begin{cases} X \\ A \\ K_1(X) \times K_2(X) \\ K_1(X) + K_2(X) \end{cases}$$

  where A is an arbitrary object of $\mathbb{C}$ and K_1 and K_2 are previously defined cartesian functors.

- An *extended cartesian functor* is an extended polynomial functor that has only cartesian functors on the left hand side of $\Rightarrow$. More precisely: A functor $G : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ is called an extended cartesian functor, if it is defined as one of the cases

$$G(Y, X) \quad = \quad \begin{cases} X \\ A \\ G_1(Y, X) \times G_2(Y, X) \\ G_1(Y, X) + G_2(Y, X) \\ K(Y) \Rightarrow G_1(Y, X) \end{cases}$$

  where A is an arbitrary object of $\mathbb{C}$, K is a cartesian functor, and G_1 and G_2 are previously defined extended cartesian functors.

In (Hermida and Jacobs, 1998; Poll and Zwanenburg, 2001) the term polynomial functor is used for what I call cartesian functor here. The difference between extended cartesian functors and extended polynomial functors is that extended cartesian functors can model only those binary methods that take no arguments with functional type (see Table 1.1 on page 6). An example for an extended cartesian functor is

$$(Y, X) \longmapsto (Y \Rightarrow A) \ + \ (X \times A) \qquad (*)$$

which corresponds to a binary method of type $\mathsf{Self} \longrightarrow (\mathsf{Self} \Rightarrow A) \ + \ (\mathsf{Self} \times A)$. The functor $\mathsf{NeighbourhoodPointIface}$ from Example 3.2.4 (on page 80) is not an extended cartesian functor, because the method $\mathsf{move\_with\_neighbour}$ takes an argument of the type $\mathsf{Addr} \Rightarrow \mathsf{Self}$.

Note that (extended) cartesian functors are a proper subclass of (extended) polynomial functors. Cartesian functors and extended cartesian functors inherit the definitions for coalgebra, coalgebra morphism, predicate and relation lifting, bisimulation and invariant from polynomial and extended polynomial functors, respectively.

Recall that a relation R is a *partial equivalence relation* if it is symmetric and transitive, that is, if $R^{\mathrm{op}} \subseteq R$ and $R \circ R \subseteq R$. Recall further that R is an *equivalence relation* if it is reflexive (i.e., $\mathrm{Eq}(X) \subseteq R$), symmetric, and transitive. It is immediate that a partial equivalence relation R is an equivalence relation on its domain $\mathsf{dom}(R)$. Following (Rutten, 2000) I call a bisimulation that is a (partial) equivalence relation a *(partial) bisimulation equivalence.*

Poll and Zwanenburg consider in (Poll and Zwanenburg, 2001) dialgebras and prove that bisimulation equivalences for dialgebras form a complete lattice. A dialgebra with carrier X is a finite set of functions of the form

$$F_{\mathrm{IN}}(X) \longrightarrow F_{\mathrm{OUT}}(X)$$

where both F_{IN} and F_{OUT} are cartesian functors. One can use distributivity to move $\times$ over $+$ to convert every dialgebra into a pair $\langle c, a\rangle$, where c is a coalgebra for some extended cartesian functor and a is a constant in $F(X)$ for some polynomial functor F. Note that there are coalgebras for extended cartesian functors that cannot be represented as a dialgebra. An example is $(*)$ from above. The result of Poll and Zwanenburg implies that there is a subclass of extended cartesian functors for which the bisimulation equivalences form a complete lattice.

In the following I generalise the result of (Poll and Zwanenburg, 2001) and prove that partial bisimulation equivalences form a complete lattice for all coalgebras of extended cartesian functors. At the end of this subsection there is an example of a coalgebra (for an extended polynomial functor) for which there is no greatest bisimulation equivalence (demonstrating that a further generalisation is impossible). The work of Poll and Zwanenburg and personal discussions with Erik Poll had a big influence on what I present here.

All the following lemmas and propositions are towards Theorem 3.5.9 (on page 113 below). All proofs have been formalised in PVS as far as this is feasible, see Section 2.4.4 and Appendix A. I start with basic properties that deal with technical side conditions.

Let me first collect some basic facts about partial equivalence relations. For a relation $R \subseteq X \times X$ the *least partial equivalence relation* containing R is denoted by $\overline{R}$. The partial equivalence relations on a set X form a complete lattice. Taking the least partial equivalence relation is a closure operation, that is $R \subseteq \overline{R}$, $R \subseteq S$ implies $\overline{R} \subseteq \overline{S}$, and $\overline{\overline{R}} = \overline{R}$. In particular $R \subseteq S$ implies $\overline{R} \subseteq S$ if S is a partial equivalence relations.

One obtains $\overline{R}$ from R by taking the symmetric and transitive closure of R. The symmetric and transitive closure can also be described as follows: Call a finite sequence

$x_1, x_2, \ldots, x_n$ (with $n \geqslant 2$) a *zigzag* in R if for all $k < n$ either $x_k \, R \, x_{k+1}$ or $x_{k+1} \, R \, x_k$. Two elements $x, x' \in X$ are related by $\overline{R}$ if and only if there is a zigzag $x, \ldots, x'$ in R. I use zigzags quite often in the following. Instead of proving $x \, \overline{R} \, x'$ I usually construct a zigzag $x, \ldots, x'$ in R.

If R_1 and R_2 are two partially reflexive relations with equal domains (in particular if R_1 and R_2 are partial equivalence relations on the same domain) then $a \, R_1 \, b$ implies both $a \, R_2 \, a$ and $b \, R_2 \, b$. For an arbitrary collection $(R_i)_{i \in I}$ of partially reflexive relations also their union $\bigvee_i R_i$ is partially reflexive.

Lemma 3.5.2 *Let $S \subseteq Y \times Y$ and $R \subseteq X \times X$ be partial equivalence relations.*

1. *The cartesian closed structure of* **Rel** *preserves partial equivalence relations, that is all of $S \times_{\mathrm{R}} R$, $S +_{\mathrm{R}} R$, and $S \Rightarrow_{\mathrm{R}} R$ are partial equivalence relations.*
2. *The relation lifting* $\mathrm{Rel}(H)(S, R)$ *is a partial equivalence relation for all higher-order polynomial functors H.*

Proof Point (1) has been proved in PVS. The only thing that is not immediate is transitivity of $S \Rightarrow_{\mathrm{R}} R$: Assume $f \, (S \Rightarrow_{\mathrm{R}} R) \, g$ and $g \, (S \Rightarrow_{\mathrm{R}} R) \, h$, we have to show that $f \, (S \Rightarrow_{\mathrm{R}} R) \, h$, that is, that $a \, S \, b$ implies $(f \, a) \, R \, (h \, b)$. From the assumptions we get $(f \, a) \, R \, (g \, b)$ and because $b \, S \, b$ also $(g \, b) \, R \, (h \, b)$ and the result follows from the transitivity of R.

Item (2) is proved by induction on the structure of H, using Item (1). □

Lemma 3.5.3 *For polynomial functors the relation lifting preserves the property of 'having the same domain'. Precisely, let $R_1, R_2 \subseteq X \times X$ be relations with* $\mathsf{dom}(R_1) = \mathsf{dom}(R_2)$*. If R_1 and R_2 are partially reflexive then*

$$\mathsf{dom}(\mathrm{Rel}(F)(R_1)) \quad = \quad \mathsf{dom}(\mathrm{Rel}(F)(R_2))$$

for all polynomial functors F.

Proof Let $R_1, R_2 \subseteq X \times X$ and $S_1, S_2 \subseteq Y \times Y$ be partially reflexive relations such that $\mathsf{dom}(R_1) = \mathsf{dom}(R_2)$ and $\mathsf{dom}(S_1) = \mathsf{dom}(S_2)$. First one proofs the following equations.

$$\begin{array}{rcl} \mathsf{dom}(S_1 \times_{\mathrm{R}} R_1) & = & \mathsf{dom}(S_2 \times_{\mathrm{R}} R_2) \\ \mathsf{dom}(S_1 +_{\mathrm{R}} R_1) & = & \mathsf{dom}(S_2 +_{\mathrm{R}} R_2) \\ \mathsf{dom}(\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_1) & = & \mathsf{dom}(\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_2) \end{array}$$

Then the lemma follows by induction on the structure of F. The above equations have been proved in PVS. □

The preceding lemma does not hold for extended polynomial functors.

Lemma 3.5.4 *Let $S \subseteq Y \times Y$ and $R \subseteq X \times X$ be relations.*

$$\overline{S +_{\mathrm{R}} R} \quad = \quad \overline{S} +_{\mathrm{R}} \overline{R} \tag{1}$$

And under the condition that both S and R are partially reflexive

$$\overline{S \times_{\mathrm{R}} R} \quad = \quad \overline{S} \times_{\mathrm{R}} \overline{R} \tag{2}$$

Proof This lemma has been proved in PVS. Let me first do $\overline{S +_{\mathrm{R}} R} \subseteq \overline{S} +_{\mathrm{R}} \overline{R}$. Because $\overline{(-)}$ is a closure operator and $\overline{S} +_{\mathrm{R}} \overline{R}$ is a partial equivalence relation by Lemma 3.5.2 (1), it is sufficient to show $S +_{\mathrm{R}} R \subseteq \overline{S} +_{\mathrm{R}} \overline{R}$, but this follows from monotonicity of $\overline{(-)}$ and $+_{\mathrm{R}}$ (Lemma 2.4.5 (2)).

For $\overline{S} +_{\mathrm{R}} \overline{R} \subseteq \overline{S +_{\mathrm{R}} R}$ assume $(\kappa_1 y,\ \kappa_1 y') \in \overline{S} +_{\mathrm{R}} \overline{R}$, so there is a zigzag $y, y_2, \ldots, y_n, y'$ in S. Then $\kappa_1 y, \kappa_1 y_2, \ldots, \kappa_1 y_n, \kappa_1 y'$ is a zigzag in $S +_{\mathrm{R}} R$, so $(\kappa_1 y,\ \kappa_1 y') \in \overline{S +_{\mathrm{R}} R}$. Similarly for the second injection.

$\overline{S \times_{\mathrm{R}} R} \subseteq \overline{S} \times_{\mathrm{R}} \overline{R}$ follows again from monotonicity and Lemma 3.5.2 (1).

For $\overline{S} \times_{\mathrm{R}} \overline{R} \subseteq \overline{S \times_{\mathrm{R}} R}$ assume $((y, x), (y', x')) \in \overline{S} \times_{\mathrm{R}} \overline{R}$, so there is a zigzag $y, y_2, \ldots, y_n, y'$ in S and a zigzag $x, x_2, \ldots, x_n, x'$ in R. Both R and S are partially reflexive, therefore (y, x), (y_2, x), $\ldots$, (y_n, x) , (y', x), (y', x_2) , $\ldots$, (y', x_n), (y', x') is a zigzag in $S \times_{\mathrm{R}} R$. So $((y, x), (y', x')) \in \overline{S \times_{\mathrm{R}} R}$. □

For the exponent one can infer

$$\overline{S \Rightarrow_{\mathrm{R}} R} \quad \subseteq \quad \overline{S} \Rightarrow_{\mathrm{R}} \overline{R}$$

under the condition that S is partially reflexive and that R is symmetric. And

$$\overline{S} \Rightarrow_{\mathrm{R}} \overline{R} \quad \subseteq \quad \overline{S \Rightarrow_{\mathrm{R}} R}$$

holds if R is a partial equivalence relation. However, these two properties do not help in the following.

Lemma 3.5.5 *Let $S_i \subseteq Y \times Y$ and $R_i \subseteq X \times X$ be two indexed families of relations for an arbitrary index set I. Assume that all the R_i and all the S_i have pairwise equal domains. Then*

$$\overline{\bigvee_i (S_i +_{\mathrm{R}} R_i)} \quad = \quad \overline{(\bigvee_i S_i) +_{\mathrm{R}} (\bigvee_i R_i)} \tag{1}$$

And under the condition that all the S_i and all the R_i are partially reflexive it holds that

$$\overline{\bigvee_i (S_i \times_{\mathrm{R}} R_i)} \quad = \quad \overline{(\bigvee_i S_i) \times_{\mathrm{R}} (\bigvee_i R_i)} \tag{2}$$

Proof Equation (1) is mainly included for completeness here, it follows immediately from Lemma 2.4.10 (2). The same applies to the inclusion from left to right in Equation (2). It remains to prove that $\overline{(\bigvee_i S_i) \times_{\mathrm{R}} (\bigvee_i R_i)} \subseteq \overline{\bigvee_i (S_i \times_{\mathrm{R}} R_i)}$. For that it is sufficient to show that $(\bigvee_i S_i) \times_{\mathrm{R}} (\bigvee_i R_i) \subseteq \overline{\bigvee_i (S_i \times_{\mathrm{R}} R_i)}$, so assume $y \, S_i \, y'$ and $x \, R_j \, x'$ for some $i, j \in I$. Now the assumptions imply that $x \, R_i \, x$ and $y' \, S_j \, y'$, so (y, x), (y', x), (y', x') is a zigzag in $\bigvee_i (S_i \times_{\mathrm{R}} R_i)$, thus $((y, x), (y', x')) \in \overline{\bigvee_i (S_i \times_{\mathrm{R}} R_i)}$. □

Proposition 3.5.6 *Let K be a cartesian functor and $(R_i)_{i \in I}$ be a family of partial equivalence relations that have pairwise equal domains. Then it holds that*

$$\mathrm{Rel}(K)(\overline{\textstyle\bigvee_i R_i}) \quad = \quad \overline{\textstyle\bigvee_i \mathrm{Rel}(K)(R_i)}$$

Proof The proof goes by induction on the structure of K. The cases of constants and identity are immediate. With the previous utility lemmas the proof for product and coproduct is almost identical. So let me do the case $K = K_1 \times K_2$ here. Note that Lemma 3.5.3 implies that $\mathsf{dom}(\mathrm{Rel}(K_{1/2})(R_i) = \mathsf{dom}(\mathrm{Rel}(K_{1/2})(R_j)$ for all $i, j \in I$. Further, Lemma 3.5.2 shows that $\mathrm{Rel}(K_{1/2})(R_i)$ is a partial equivalence relation for all i, so $\bigvee_i \mathrm{Rel}(K_{1/2})(R_i)$ is partially reflexive.

$$\begin{array}{rcll}
\mathrm{Rel}(K_1 \times K_2)(\overline{\bigvee_i R_i}) & = & \mathrm{Rel}(K_1)(\overline{\bigvee_i R_i}) \;\times_{\mathrm{R}}\; \mathrm{Rel}(K_2)(\overline{\bigvee_i R_i}) & \\
& = & \overline{\bigvee_i \mathrm{Rel}(K_1)(R_i)} \;\times_{\mathrm{R}}\; \overline{\bigvee_i \mathrm{Rel}(K_2)(R_i)} & \text{by Ind. Hyp.} \\
& = & \overline{\bigvee_i \mathrm{Rel}(K_1)(R_i) \;\times_{\mathrm{R}}\; \bigvee_i \mathrm{Rel}(K_2)(R_i)} & \text{by 3.5.4 (2)} \\
& = & \overline{\bigvee_i \big(\mathrm{Rel}(K_1)(R_i) \;\times_{\mathrm{R}}\; \mathrm{Rel}(K_2)(R_i)\big)} & \text{by 3.5.5 (2)} \\
& = & \overline{\bigvee_i \mathrm{Rel}(K_1 \times K_2)(R_i)} & \square
\end{array}$$

Remark 3.5.7

1. The important part of the preceding proposition is the subset relation from left to right: Assume we are given a pair $(t, t') \in \mathrm{Rel}(K)(\overline{\bigvee_i R_i})$ then we can safely assume that there is a zigzag $t, t_2, \ldots, t_n, t'$ in $\bigvee_i \mathrm{Rel}(K)(R_i)$.

2. The proposition does not hold for polynomial functors because for the exponent one can only derive

$$\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} \overline{\textstyle\bigvee_i R_i} \quad \supseteq \quad \overline{\textstyle\bigvee_i (\mathrm{Eq}(A) \Rightarrow_{\mathrm{R}} R_i)} \qquad (3.1)$$

   By induction this gives for a polynomial functor F

$$\mathrm{Rel}(F)(\overline{\textstyle\bigvee_i R_i}) \quad \supseteq \quad \overline{\textstyle\bigvee_i \mathrm{Rel}(F)(R_i)}$$

   (which follows also directly from 2.6.9 (1) and (5), 3.5.2, and from the monotonicity of $\overline{(-)}$). It is quite easy to find examples in which the subset relation in (3.1) is strict (see Example 3.5.10 below).

Lemma 3.5.8 *Let G be an extended cartesian functor and let $(S_i) \subseteq Y \times Y$ and $(R_i) \subseteq X \times X$ be two families of partial equivalence relations indexed by an arbitrary set I. Assume that all the S_i and all the R_i have pairwise equal domains. Let $(s, r) \in \mathrm{Rel}(G)(S_j, R_j)$ for some $j \in I$. If we have additionally that for all $i \in I$ both $(r, r) \in \mathrm{Rel}(G)(S_i, R_i)$ and $(s, s) \in \mathrm{Rel}(G)(S_i, R_i)$ then we have also $(s, r) \in \mathrm{Rel}(G)(\overline{\bigvee_i S_i}, \overline{\bigvee_i R_i})$.*

Proof The proof proceeds by induction on the structure of G. All the induction steps have been formalised in PVS. The cases of $G(Y, X) = A$ and $G(Y, X) = X$ are (as always) immediate. The cases of product and coproduct are proved by unpacking the definition of relation lifting and invoking the induction hypothesis.

Let me demonstrate $G(Y, X) = K(Y) \Rightarrow G_1(Y, X)$ in detail. From the assumptions it follows that s and r are functions such that there exists $j \in I$ with

$$\forall a, b \in K(Y)\,. \quad \mathrm{Rel}(K)(S_j)(a, b) \quad \text{implies} \quad \mathrm{Rel}(G_1)(S_j, R_j)(s\,a, r\,b) \tag{1}$$

and that for all $i \in I$

$$\forall a, b \in K(Y)\,. \quad \mathrm{Rel}(K)(S_i)(a, b) \quad \text{implies} \quad \mathrm{Rel}(G_1)(S_i, R_i)(s\,a, s\,b) \tag{2}$$

$$\forall a, b \in K(Y)\,. \quad \mathrm{Rel}(K)(S_i)(a, b) \quad \text{implies} \quad \mathrm{Rel}(G_1)(S_i, R_i)(r\,a, r\,b) \tag{3}$$

It remains to show that

$$\forall a, b \in K(Y)\,. \quad \mathrm{Rel}(K)(\overline{\textstyle\bigvee_i S_i})(a, b) \quad \text{implies} \quad \mathrm{Rel}(G_1)(\overline{\textstyle\bigvee_i S_i},\ \overline{\textstyle\bigvee_i R_i})(s\,a,\ r\,b)$$

So assume $(a, b) \in \mathrm{Rel}(K)(\overline{\bigvee_i S_i})$, by Proposition 3.5.6 there is a zigzag $a_1, \dots, a_n$ with $a_1 = a$ and $a_n = b$ in $\bigvee_i \mathrm{Rel}(K)(S_i)$. Build now the sequence $s\,a_1$, $s\,a_2$, … $s\,a_n$, $r\,a_n$ and invoke the induction hypothesis for G_1 on each two adjacent elements to show that it is a zigzag in $\mathrm{Rel}(G_1)(\overline{\bigvee_i S_i},\ \overline{\bigvee_i R_i})$. This completes the proof because the latter is a partial equivalence relation by Lemma 3.5.2.

Checking the assumptions of the induction hypothesis we get by Lemma 3.5.2 that $\mathrm{Rel}(K)(S_i)$ is partially reflexive thus by assumption (2) $(s\,a_k, s\,a_k) \in \mathrm{Rel}(G_1)(S_i, R_i)$ for all $k \leq n$ and all $i \in I$. Because $a_1, \dots, a_n$ is a zigzag we know that for each $k < n$ there is some $j \in I$ such that either (a_k, a_{k+1}) or (a_{k+1}, a_k) is contained in $\mathrm{Rel}(K)(S_j)$, so by assumption (2) we have that either $(s\,a_k, s\,a_{k+1})$ or $(s\,a_{k+1}, s\,a_k)$ is contained in $\mathrm{Rel}(G_1)(S_j, R_j)$. Similarly (1) dispatches the assumptions for invoking the induction hypothesis on the pair $(s\,a_n,\ r\,a_n)$. □

Theorem 3.5.9 *Let $c : X \longrightarrow G(X, X)$ be a coalgebra for an extended cartesian functor G. The partial bisimulation equivalences for c on a fixed domain form a complete lattice. In this lattice the join of a family (R_i) of bisimulations is given by $\overline{\bigvee_i R_i}$. In particular $\overline{\bigvee_i R_i}$ is a bisimulation for any family (R_i) of bisimulation equivalences.*

Proof It is sufficient to show that for an arbitrary indexed family (R_i) of partial bisimulation equivalences with pairwise equal domains the relation $\overline{\bigvee_i R_i}$ is a bisimulation. If R_i is a bisimulation for c then $(c, c) \in \mathrm{Rel}(X \Rightarrow G)(R_i, R_i)$. Invoking Lemma 3.5.8 on the collection (R_i) and the pair (c, c) yields then the required $(c, c) \in \mathrm{Rel}(X \Rightarrow G)(\overline{\bigvee_i R_i}, \overline{\bigvee_i R_i})$. □

Example 3.5.10 This is the announced example of a coalgebra for which there is an infinite ascending chain of bisimulation equivalences such that every upper bound of this chain is not a bisimulation. This example shows that the preceding theorem cannot be generalised to all extended polynomial functors. Consider the extended polynomial functor

$$K(Y, X) \quad \stackrel{\mathrm{def}}{=} \quad (\mathbb{N} \Rightarrow Y) \Rightarrow \mathsf{bool}$$

For a function $g : V \longrightarrow Y$ (the covariant argument is ignored) the action on morphisms is

$$\begin{array}{rcl} K(g, X) & : & (\mathbb{N} \Rightarrow Y) \Rightarrow \mathsf{bool} \longrightarrow (\mathbb{N} \Rightarrow V) \Rightarrow \mathsf{bool} \\ & = & \lambda h \,.\, \lambda k \,.\, h(g \circ k) \end{array}$$

To give a readable presentation of the relation lifting for K I need the notion of R–related functions for a relation $R \subseteq U \times V$. Two functions $a : \mathbb{N} \longrightarrow U$ and $b : \mathbb{N} \longrightarrow V$ are R–related if $R(a\,n, b\,n)$ holds for all $n \in \mathbb{N}$. The relation lifting for K is

$$\begin{array}{rcl} \mathrm{Rel}(K)(R, -) & \subseteq & K(U, -) \times K(V, -) \\ & = & \big\{(f, g) \mid f : (\mathbb{N} \Rightarrow U) \longrightarrow \mathsf{bool}, g : (\mathbb{N} \Rightarrow V) \longrightarrow \mathsf{bool} \;\; \text{such that} \\ & & \quad \text{for all } R\text{–related functions } a \text{ and } b : f\,a = g\,b\big\} \end{array}$$

(I omitted the covariant argument because it is ignored anyway.)

In the following I define a K–coalgebra c on state space $\mathbb{N}$. So c takes as second argument a function $\mathbb{N} \longrightarrow \mathbb{N}$. I call such a function *bounded* if there exists a natural number n such that $f\,i < n$ for all $i \in \mathbb{N}$. The main point is, that for the total relation $\top_{\mathbb{N} \times \mathbb{N}}$ there exist pairs of functions $a, b : \mathbb{N} \longrightarrow \mathbb{N}$ such that a and b are $\top_{\mathbb{N} \times \mathbb{N}}$–related and a is bounded while b is not bounded. The coalgebra c is now defined as

$$\begin{array}{rcl} c & : & \mathbb{N} \longrightarrow (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow \mathsf{bool} \\ & = & \lambda i \,.\, \lambda a \,.\, \begin{cases} \top & a \text{ is bounded} \\ \bot & \text{otherwise} \end{cases} \end{array}$$

where $\top$ and $\bot$ are the two elements of bool. The construction of c ensures that the total relation $\top_{\mathbb{N} \times \mathbb{N}}$ is not a bisimulation for c. Consider now the family $(S_n)_{n \in \mathbb{N}}$ of relations defined by

$$i\, S_n\, j \quad \text{if and only if} \quad i = j \;\text{ or }\; (i < n \;\wedge\; j < n)$$

For any of the S_n and two functions a and b the following holds: If a and b are S_n related then a is bounded precisely when b is. Therefore all the S_n are bisimulation equivalences. Further, in accordance with Proposition 3.4.30, any finite set of relations S_n has an upper bound, which is a bisimulation for c. The least upper bound of all S_n is the total relation, so there is no greatest bisimulation equivalence.

In (Tews, 2002b) I further investigate the problem of the existence of greatest bisimulation equivalences for extended polynomial functors. There I show that for *finitely based* coalgebras a greatest bisimulation equivalence does always exist. A restriction to finitely based coalgebras is not unreasonable: It essentially excludes those coalgebras that are not computable. ■

3.6. Final Coalgebras for Generalised Polynomial Functors

In this Section I discuss the existence of final coalgebras for generalised polynomial functors. Recall that an object z in a category $\mathbb{C}$ is a final object, if for any object x in $\mathbb{C}$ there exists exactly one morphism $x \longrightarrow z$. This unique morphism is usually denoted with $!_x$. Final coalgebras (in a suitable category of coalgebras) are minimal (any bisimulation for the final coalgebra is contained in the equality relation) and they realize all possible behaviours (because any other coalgebra can be embedded into the final one via the unique morphism !). Final coalgebras can give semantics to behavioural types or class specifications (Reichel, 1995).

For all bounded functors[7] (including all polynomial functors) a final coalgebra does exist (Kawahara and Mori, 2000). So it is natural to ask if those extended cartesian (or extended polynomial) functors, which are not equivalent to polynomial functors, have final coalgebras. I already said above, that the functor T from Example 3.3.8 has a final coalgebra. This is the case because one cannot make any observations from a T–coalgebra.

More general, let H be a higher-order polynomial functor such that $H(\mathbf{1}, \mathbf{1})$ is isomorphic to $\mathbf{1}$. Then H has a final coalgebra, it is the only function $\mathbf{1} \longrightarrow H(\mathbf{1}, \mathbf{1})$.

Consider now an extended cartesian functor G that does allow for an observation (i.e., $G(\mathbf{1}, \mathbf{1}) \not\cong \mathbf{1}$). If G is not naturally isomorphic to a polynomial functor, that is, if G models a signature that contains at least one binary method, then one can use a diagonalisation argument to show that G cannot have a final coalgebra. In the following I prove this claim only for one particular extended cartesian functor. However, the proof generalises to all such extended cartesian and extended polynomial functors.

[7] An endofunctor T is bounded if there is a cardinality ρ such that for every T–coalgebra $X \longrightarrow T(X)$ and for every $x \in X$ the least invariant containing x is bounded by ρ, see (Kawahara and Mori, 2000).

Proposition 3.6.1 *The functor $G(Y, X) = (Y \Rightarrow \mathsf{bool})$ does not admit a final coalgebra.*

(The functor $G(Y, X) = (Y \Rightarrow \mathsf{bool})$ corresponds to a signature with one binary method $\mathsf{m} : \mathsf{Self} \times \mathsf{Self} \longrightarrow \mathsf{bool}$.)

Proof This proposition has been proved in PVS. Let me denote the two elements of bool with t and f and let $\mathsf{not} : \mathsf{bool} \longrightarrow \mathsf{bool}$ be the function, which maps t to f and vice versa.

Assume towards a contradiction that $z : Z \longrightarrow Z \Rightarrow \mathsf{bool}$ is a final coalgebra. Construct a new coalgebra $z' : (Z + 1) \longrightarrow (Z + 1) \Rightarrow \mathsf{bool}$ by

$$z'(x)(y) = \begin{cases} z(x')(y') & \text{if } x = \kappa_1\, x' \text{ and } y = \kappa_1\, y' \\ \mathsf{not}(z(x')(x')) & \text{if } x = \kappa_1\, x' \text{ and } y = \kappa_2 * \\ \mathsf{t} & \text{otherwise.} \end{cases}$$

By construction $\kappa_1 : Z \longrightarrow Z + 1$ is a coalgebra morphism $z \longrightarrow z'$. Because z is the final coalgebra, there exists $!_{z'} : Z + 1 \longrightarrow Z$. Because $!_{z'}$ is a coalgebra morphism $z' \longrightarrow z$ the equation $z'\, x\, y = z\, (!_{z'}\, x)\, (!_{z'}\, y)$ holds for all $x, y \in Z + 1$. Further, we have $!_{z'} \circ \kappa_1 = \mathsf{id}_Z$ because there can only be one coalgebra morphism $z \longrightarrow z$. Let $*$ be the only element of $\mathbf{1}$ and let $\boxast\ = !_{z'}(\kappa_2 *)$. Then

$$\begin{array}{rcll} \mathsf{not}(z(\boxast)(\boxast)) & = & z'(\kappa_1 \boxast)(\kappa_2 *) & \text{Definition of } z' \\ & = & z(!_{z'}(\kappa_1 \boxast))(\boxast) & \text{Apply } !_{z'} : z' \longrightarrow z \\ & = & z(\boxast)(\boxast) & \text{Use } !_{z'} \circ \kappa_1 = \mathsf{id}_Z \end{array}$$

which is clearly impossible. Thus, there is no final coalgebra for G. □

The preceding result has serious consequences. For coalgebras of polynomial endofunctors there exists a *coinduction principle*, see Subsection 2.6.7. It states that the bisimilarity relation for the final coalgebra is contained in the equality relation. For extended polynomial and extended cartesian functors this principle is vacuous. First, the final coalgebra does not exist for interesting functors of these classes. Second, for extended polynomial functors, it is unclear what bisimilarity should be (because there is no greatest bisimulation).

A notion of coalgebra without a corresponding notion of coinduction loses much of its original attraction. However, I would like to argue here, that an important application area of coalgebras is object-oriented specification. Without an appropriate treatment of binary methods, coalgebraic specification will remain an exotic area in theoretical computer science. With all the problems that were caused by binary methods in the past, it is unrealistic to hope that one can get binary methods for free.

Apart from the coinduction principle, the existence of a final model can serve two purposes. First, it tells the person who developed the specification that the specification is itself consistent. Second, he can examine the states of the final model and if he does not find any unwanted behaviour, he can be sure that the specification captures the right class of models. Therefore I would like to propose that for any coalgebraic specification the developer should convince himself that the final model exists. Once he did this, he can also use an appropriate coinduction principle (even for signatures that correspond to higher-order polynomial functors).

The question of sufficient conditions for the existence of a final coalgebra (in a given class of coalgebras) remains open for future research. For polynomial functors Kurz recently solved this problem in (Kurz, 2002). In this work Kurz characterises those coalgebraic logics that admit final semantics. A logic admits final semantics if every definable class of coalgebras contains a final coalgebra.

For the present thesis I use the construction of the final model as described in (Jacobs, 1996b). For class specifications in CCSL I give in Proposition 4.5.18 a sufficient condition that ensures that the final model constructed this way is nonempty (see also Theorem 4.7.4 on page 206).

3.7. Summary

This chapter generalises the familiar notion of coalgebra to a form such that coalgebras can model arbitrary class signatures built up from the polynomial type constructions $+, \times, \Rightarrow$ and from constants. This includes in particular arbitrary binary methods. The framework presented here is more general than what would be needed to reason about Java or Eiffel. In both languages it is not possible to form function types, so in both languages one can declare the method equal from the examples of points (page 74) but not the method move_with_neighbours (page 75). For the verification of Java and Eiffel one can restrict oneself to the class of extended cartesian functors.

The generalisation of coalgebras works by defining three new classes of functors: *Higher-order polynomial functors* that can model arbitrary signatures, *extended polynomial functors* that can model signatures with a restricted use of the exponential $\Rightarrow$, and *extended cartesian functors* for even more restricted signatures (see Table 1.1 on page 6 for a comparison). For these three classes of functors I gave definitions for the notions of coalgebra, coalgebra morphism, bisimulation, and invariant. All these definitions generalise the corresponding notions for polynomial functors: every polynomial functor is an extended cartesian functor, every extended cartesian functor is an extended polynomial functor, and every extended polynomial functor is a higher-order polynomial functor. Further, if $\alpha : X \longrightarrow F(X)$ is a coalgebra for a polynomial functor F then it is also a coalgebra for F seen as one of the more general functors. The notions of bisimulation and invariant for α are the same, regardless whether one considers F as a higher-order, extended polynomial, extended cartesian, or traditional polynomial functor.

The results about coalgebras for polynomial functors that have been collected from the literature in Section 2.6 serve as a benchmark for the proposed generalisation of the notion of coalgebra. Higher-order polynomial functors turn out to be too expressive to have many useful properties: Almost all familiar results that are known to hold for coalgebras of polynomial functors do not hold for coalgebras of higher-order polynomial functors. For most of these negative results this chapter presents counter examples, thus proving that these properties do not hold in general. For reasons of space some counter examples have been omitted, for instance an example of a coalgebra morphism whose kernel is not a bisimulation. These omitted counter examples are part of the PVS formalisation of this thesis and are available on the world wide web, see Appendix A.

Extended polynomial functors, which are a proper subclass of higher-order polynomial functors, behave much nicer. With three exceptions Section 3.4 proves all the results from the introduction on coalgebras (Section 2.6) for extended polynomial functors. The three exceptions are the following: the existence of a final coalgebra and union of invariants and bisimulations. As a consequence, the greatest bisimulation and the greatest invariant contained in some predicate do not exist for extended polynomial functors.

Concerning the three exceptions there is the following to note: Proposition 3.6.1 shows that there is no way to recover the existence of final coalgebras for extended cartesian functors. However, in applications one often restricts the behaviour of the coalgebra. It is often the case that in such a restricted class of coalgebras a final coalgebra *does* exist.

Concerning the union of invariants, the Example 3.3.4 shows that the notion of invariant according to Definition 3.3.3 might not capture everybody's intuition about invariance. The alternative, the strong invariants presented in Subsection 3.4.6 are closed under arbitrary union. So for defining a logic for coalgebras the strong invariants are more appropriate because strong invariants can give semantics to the modal operators always and eventually, see Subsection 4.5.2. However, there are problems in the interplay between strong invariants and bisimulations. Only some of the propositions relating bisimulations and invariants do hold for strong invariants.

For the union of bisimulations the solution is to strengthen the definition of bisimulation. If bisimulations are required to be partially reflexive then there is an upper bound for any two bisimulations. If one requires that bisimulations are partial equivalence relations then, for the class of extended cartesian functors, bisimulations form a complete lattice. For the class of extended polynomial functors a similar result can be obtained by placing restrictions on the coalgebra, see (Tews, 2002b). One direction for future work is the generalisation of Theorem 3.5.9 to bisimulations between two different coalgebras. This could be done along the lines of Proposition 3.4.31.

So in summary it is fair to say that coalgebras for extended polynomial functors solve the longstanding problem of incorporating binary methods into coalgebraic specification.

The approach to use extended polynomial functors is more general than other approaches that allow one to model binary methods. In comparison with the use of definitional extensions and algebraic extensions (see page 78f.) extended polynomial functors allow the binary method to contribute to the observable behaviour. Further the result type of binary methods is not restricted to Self.

In comparison with hidden algebra (Roşu, 2000; Goguen and Malcolm, 2000) extended polynomial functors offer more flexibility with respect to the types of the methods (recall that in hidden algebra operations are restricted to the type $S_1 \times \cdots \times S_n \longrightarrow S_0$, where all the S_i are sorts). Additionally, the coalgebraic approach offers the notion of bisimulation to compare states of *different* models. In case binary methods are present one can only compare states of one model with hidden congruences. In this case the precise relation of hidden congruences and bisimulations (on one coalgebra) is unclear. If a (single sorted) hidden signature contains no binary method then it corresponds to a polynomial functor. Under this restriction the greatest hidden congruence corresponds to bisimilarity.

An important problem that has not been investigated in the present thesis is the interplay of extended polynomial functors (or higher-order polynomial functors) with the (iterated) data functors from (Hensel, 1999). The class of data functors is an extension of polynomial functors with least and greatest fixed points. For instance from the functor $T_{\mathsf{List}}(A, X) = A \times X + \mathbf{1}$ one obtains as least fixed point the functor $\mathsf{List}(A) = \mu X \,.\, T_{\mathsf{List}}(A, X)$ that sends a set A to the set of lists over A. The greatest fixed point is $\mathsf{Seq}(A) = \nu X \,.\, T_{\mathsf{List}}(A, X)$, the functor that gives (possibly infinite) sequences. The construction can be continued, one obtains possibly infinitely branching trees of finite depth as $A \longmapsto \mu X \,.\, \mathsf{Seq}(A \times X)$. (Hensel, 1999) also describes the predicate and relation lifting for data functors. In the case of lists this is

$$\begin{aligned} \mathrm{Pred}(\mathsf{List})(P) &\subseteq \mathsf{List}(X) \\ \mathrm{Rel}(\mathsf{List})(R) &\subseteq \mathsf{List}(X) \times \mathsf{List}(Y) \end{aligned}$$

for a predicate $P \subseteq X$ and a relation $R \subseteq X \times Y$, characterised by

$$\mathrm{Pred}(\mathsf{List})(P)(l) \quad \text{if and only if} \quad \begin{cases} l = \mathsf{nil} & \text{or} \\ l = \mathsf{cons}(x, l') \ \wedge \ P(x) \ \wedge \ \mathrm{Pred}(\mathsf{List})(P)(l') \end{cases}$$

$$\mathrm{Rel}(\mathsf{List})(R)(l_1, l_2) \quad \text{if and only if} \quad \begin{cases} l_1 = \mathsf{nil} \ \wedge \ l_2 = \mathsf{nil} \qquad \text{or} \\ l_1 = \mathsf{cons}(x, l_1') \ \wedge \ l_2 = \mathsf{cons}(y, l_2') \ \wedge \\ \qquad\qquad R(x, y) \ \wedge \ \mathrm{Rel}(\mathsf{List})(R)(l_1', l_2') \end{cases}$$

The question is, if one can allow arbitrary data functors as ingredient functors of extended polynomial functors. (It does not make sense to take fixed points of proper extended polynomial functors, because neither initial algebras nor final coalgebras exist for proper extended polynomial functors.) If data functors are allowed, one can have a method declaration

$$m \quad : \quad \mathsf{Self} \times \mathsf{List}[\mathsf{Self}] \longrightarrow \mathsf{Self} \times A$$

that corresponds to a functor $T : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}$ given by

$$T(Y, X) \quad = \quad \mathsf{List}(Y) \Rightarrow \mathsf{Self} \times A \qquad (*)$$

The method m behaves similarly to a binary method and the functor T shares properties with extended cartesian functors. For instance there is no final coalgebra for T and T–bisimulations and T–invariants are closed under intersection but not under union. Closure under union can be obtained by considering partial bisimulation equivalences only. All this hinges on the following properties of predicate and relation lifting for lists.

Lemma 3.7.1

1. *Predicate and relation lifting for lists is monotone.*

2. *Predicate and relation lifting for lists is both fibred and cofibred.*

3. *The following commutation properties hold*

$$\begin{array}{rcll}
\mathrm{Pred}(\mathsf{List})(\top_X) & = & \top_{\mathsf{List}(X)} & \\
\mathrm{Rel}(\mathsf{List})(\mathrm{Eq}(X)) & = & \mathrm{Eq}(\mathsf{List}(X)) & \\
(\mathrm{Rel}(\mathsf{List})(R))^{\mathrm{op}} & = & \mathrm{Rel}(\mathsf{List})(R^{\mathrm{op}}) & \\
\mathrm{Rel}(\mathsf{List})(R) \circ \mathrm{Rel}(\mathsf{List})(S) & = & \mathrm{Rel}(\mathsf{List})(R \circ S) & \\
\bigwedge_i \mathrm{Pred}(\mathsf{List})(P_i) & = & \mathrm{Pred}(\mathsf{List})(\bigwedge_i P_i) & \\
\bigvee_i \mathrm{Pred}(\mathsf{List})(P_i) & \subseteq & \mathrm{Pred}(\mathsf{List})(\bigvee_i P_i) & \\
\bigwedge_i \mathrm{Rel}(\mathsf{List})(R_i) & = & \mathrm{Rel}(\mathsf{List})(\bigwedge_i R_i) & (\text{for } I \neq \emptyset) \\
\bigvee_i \mathrm{Rel}(\mathsf{List})(R_i) & \subseteq & \mathrm{Rel}(\mathsf{List})(\bigvee_i R_i) & \\
\coprod_{\pi_1} \mathrm{Rel}(\mathsf{List})(R) & = & \mathrm{Pred}(\mathsf{List})(\coprod_{\pi_1} R) & \\
\mathrm{Rel}(\mathsf{List})(R) \wedge \pi_1^* \mathrm{Pred}(\mathsf{List})(P) & = & \mathrm{Rel}(\mathsf{List})(R) \wedge \mathrm{Rel}(\mathsf{List})(\pi_1^* P) &
\end{array}$$

for arbitrary predicates and relations.

4. *If R is partially reflexive then*

$$\overline{\mathrm{Rel}(\mathsf{List})(R)} \quad = \quad \mathrm{Rel}(\mathsf{List})(\overline{R})$$

And under the assumption that (R_i) is a family of partial reflexive relations on the same domain:

$$\overline{\bigvee_i \mathrm{Rel}(\mathsf{List})(R_i)} \quad = \quad \overline{\mathrm{Rel}(\mathsf{List})(\bigvee_i R_i)}$$

Proof The proofs are trivial or by induction on the structure of lists. Everything has been proved in PVS. □

This shows that one can allow the list construction in extended polynomial functors and also in cartesian functors: All the results from the preceding section remain valid under this generalisation. So one can conjecture that extended polynomial functors keep their properties if one allows arbitrary data functors as ingredients. For the result about bisimulation equivalences Example 3.5.10 shows that one cannot allow the sequence functor Seq as ingredient of cartesian functors. I conjecture that extended cartesian functors keep their properties if one allows least fixed points and iterated least fixed points as ingredients of cartesian functors.

Another direction of generalisation that remains for future work is the inclusion of the (covariant) powerset functor to model nondeterminism.

4. The Specification Language CCSL

This chapter describes the *Coalgebraic Class Specification Language* CCSL. The most distinguishing feature of CCSL is the provided notion of coalgebraic specification. Further, CCSL does not force its users into a religious decision to adopt either the algebraic or the coalgebraic point of view. Instead CCSL encourages the combination of abstract data type specifications with coalgebraic specifications in an iterative way. Real world examples often involve both: abstract data types *and* behavioural aspects (or process types). Such examples can be mapped to CCSL in a very natural way. CCSL was first presented to public in (Hensel et al., 1998), a recent reference is (Rothe et al., 2001), and some more technical aspects are described in (Tews, 2002a).

The specification language CCSL (together with some supporting tools) was developed in close cooperation with the people who are associated within the LOOP project on formal methods for object-oriented programming. LOOP stands for *Logic for Object-Oriented Programming*; see the introduction in Chapter 1 for more information about the LOOP project.

CCSL is based on the observation of (Reichel, 1995) that coalgebras can give semantics to classes of object-oriented languages. Jacobs picked this idea up and developed it further in a series of publications, see (Jacobs, 1995; Jacobs, 1996b; Jacobs, 1997a; Jacobs, 1997b). Some important notions (for instance that of an *invariant*) and even parts of the syntax of CCSL can be traced back to this work. An important difference between this earlier work of Jacobs on coalgebraic specification and the work in the LOOP project is, that all work in the LOOP project is centred around *mechanical verification*. There are two reasons for the shift towards mechanical verification. First, software verification is intrinsically difficult because it involves a large amount of detail, especially many case distinctions. So to apply software verification to real programs written in a mainstream programming language (as opposed to academic examples written in a clean academic programming language) requires tool support. With the right computer support, the person who carries out the verification can concentrate on the important (and difficult) parts, while the verification tool carries out simple computations and ensures accuracy and the correctness of the whole verification.

Secondly also for academic environments and pure science right tool support is important. It enables the scientist to test his or her results and to get inspiration from large examples. For instance the work on coalgebraic refinement, presented in (Jacobs and Tews, 2001), was inspired by a large case study on coalgebraic specification of lists.

The design goals of CCSL are:

1. to provide a notation for parametrised class specifications based on coalgebras;
2. to provide algebraic specifications of abstract data types based on initial algebras;
3. to use a familiar logic;
4. to restrict expressiveness only when absolutely necessary;
5. to provide theorem proving support.

Let me discuss design goal 5 first. The importance of theorem proving support has been explained before. In order to provide a theorem proving environment for CCSL there is the following alternative: On the one hand one can write a special purpose theorem prover. On the other hand one can develop a front end to existing theorem provers. The former variant sounds attractive but is (and was) far beyond the man power of the LOOP team. In the LOOP project we therefore chose the latter variant. It has the additional advantage that we can choose among the available theorem provers and thus profit from the work that has been spent into these tools.

The front end that connects CCSL with a theorem prover can be seen as a compiler that translates CCSL into its semantics in higher-order logic. There exists a prototype implementation that supports the two theorem provers PVS (Owre et al., 1996; Owre et al., 1995) and ISABELLE/HOL in the new style ISAR syntax (Nipkow et al., 2002b; Wenzel, 2002) (but see also (Nipkow et al., 2002a; Paulson, 2002)). I refer to this prototype as the *CCSL compiler* from now on.

To meet design goals 1 and 2 CCSL contains concrete syntax for algebraic and for coalgebraic signatures. The concrete syntax for coalgebraic signatures uses terminology from object-oriented programming, for instance coalgebraic operations are declared with the keyword `METHOD`. With coalgebraic signatures one cannot describe the construction of new objects. Therefore class signatures in CCSL contain a degenerated algebraic signature (describing the constructors) in addition to the coalgebraic signature.

For design goal 2 CCSL currently supports only abstract data type specifications in the sense of the present thesis. That is, the abstract data types of CCSL do neither contain axioms nor equations. Both PVS and ISABELLE/HOL have extensions for the definition of abstract data types (without axioms). So it is straightforward to translate the abstract data types of CCSL into PVS and ISABELLE. To incorporate algebraic specifications into CCSL it would be necessary to define their semantics in higher-order logic. The problem here is that, although it is well known how to translate algebraic specifications into higher-order logic, this is quite a bit of work. Besides, such a translation has been done before (for instance for CASL in the common framework initiative (Mosses, 1997), see (Mossakowski, 2000)) and one cannot expect many new insights. It is very difficult to get this kind of work done in an academic environment.

A specification language comes always equipped with some kind of logic. A variety of different logics for coalgebras have been developed so far. One idea is that coalgebraic logic should be based on *coequations*, which are dualized equations. This approach is for instance pursued in (Corradini, 1998; Cîrstea, 1999). Other work proposes different modal logics, see for instance (Moss, 1999; Kurz, 2000; Rößiger, 2000a; Hughes, 2001). However, the work on modal logic for coalgebras is mostly driven by purely mathematical interests. The resulting logics are not well-suited for a specification language. The most modest approach for a coalgebraic logic considers coalgebraic signatures as special polymorphic signature and uses traditional first-order equational logic over these signatures (see for instance (Jacobs, 1996b; Kurz, 1998)). Such an equational logic is already sufficient for many examples. In the LOOP project we decided to use a higher-order equational logic to gain expressiveness.

Higher-order equational logic is certainly a well-known logic, as demanded by design goal number 3. It makes the semantics of CCSL specifications easy to understand. This way the user can devote his attention on the properties instead on how to express them.

With the choice of higher-order logic we deliberately neglected some proof theoretic issues. For instance there does not exist a complete derivation system for the logic of CCSL. In contrast, (Corradini, 1998), (Cîrstea, 1999), and also (Rößiger, 2000a) restrict their coalgebraic signatures and obtain a complete derivation system for their logics. As usual in interactive theorem proving, the lack of a completeness theorem has never been a problem in our case studies. In contrast, the additional expressivity of our notion of coalgebraic class signatures and of our logic turned out to be very useful. Similarly other famous negative results for higher-order logic, like the undecidability of unification, have never posed any problems.

The design goal number 4 is probably the most debatable one. Because of the expressiveness of CCSL an user can easily write inconsistent specifications. Further, it is possible to construct coalgebraic signatures that correspond to functors whose properties have not been investigated (yet). There are two arguments here. The first one is about correctness: Whatever the user writes in CCSL, the final working environment is either PVS or ISABELLE. Because the translation of CCSL uses almost no axioms[1] any inconsistency that passes the CCSL compiler is finally caught in the theorem prover. The bottom line here is that one can rest on the correctness of the theorem prover.

The second argument is that CCSL is a research tool that helps to explore the fascinating world of coalgebraic specification. If we exclude from CCSL everything that is not well understood then we cannot use it for future research.

Figure 4.1 depicts the working environment for coalgebraic specification when using CCSL. The CCSL compiler reads files containing coalgebraic specifications and produces output for either PVS or ISABELLE/HOL (depending on a command line switch). The produced output can be directly loaded into PVS or ISABELLE. In the following I refer to

[1] The exceptions are the axioms about the existence of loose and final models that facilitate aggregation. Subsection 4.7.3 gives guidelines on the use of the generated theories that ensure that these axioms cannot introduce inconsistencies.

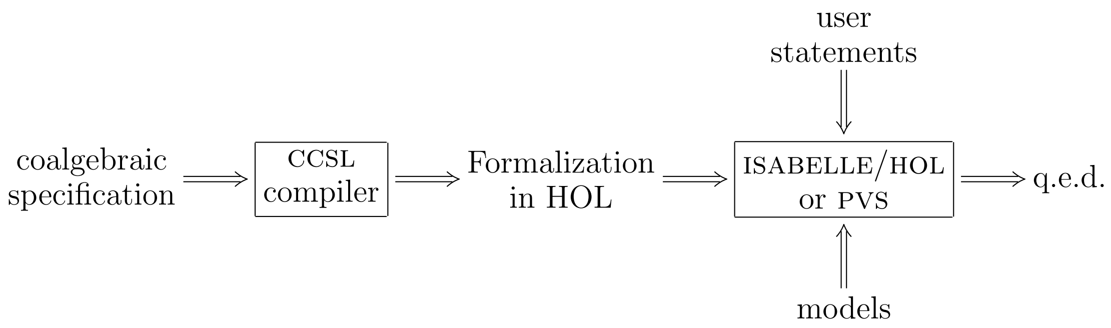


Figure 4.1.: Working environment with CCSL

the chosen proof environment as the (target) theorem prover. The formulae one wants to prove and the models are usually formulated in the logic of the target theorem prover.

Internally the CCSL-compiler uses an abstract representation of higher-order logic, so that a third theorem prover could easily be supported by adding a new pretty-printing module that translates the abstract representation into the concrete syntax of the new theorem prover. For the work on Java and JML in the LOOP project a similar compiler has been developed (van den Berg and Jacobs, 2001) that translates Java (respectively JML) into PVS or ISABELLE. Initially the compiler for Java and JML was derived from an early version of the CCSL compiler. At the moment both tools are separate programs that share parts of the internal data structures and a few modules (for instance the pretty printers for PVS and ISABELLE/HOL).

Some of the material of this chapter appeared partially already somewhere else. A simplified version of ground signatures and coalgebraic class specifications (without a proper treatment of variances) appeared originally in (Rothe et al., 2001). An even simpler version that might be called *first order coalgebraic class specification without binary methods* can be found in (Tews, 2000a). The CCSL grammar is taken from the CCSL reference manual (Tews, 2002a). In comparison with these cited papers this chapter presents the syntax and the semantics of CCSL and its coalgebraic and type theoretic foundations together. Moreover, the material is presented here without the simplifications that were necessary because of the available space and the expected audience in (Rothe et al., 2001) and (Tews, 2000a).

In describing the semantics of CCSL I am facing the following problem: The semantics of coalgebraic class specifications and that of abstract data type specifications are mutually dependent. Therefore I first describe *ground signatures* and their models. In the beginning it will be a bit unclear, where the items in the ground signature come from and how they get their semantics. After that I describe the semantics of class specifications and abstract data type specifications with respect to a given ground signature (and a model of it). In the end I close the circle with the discussion of iterated specifications:

There I show how class specifications and abstract data type specifications extend the ground signature with new types and constants.

This chapter is structured as follows: Each section introduces one syntactic category or a specific problem of CCSL. The first section is on types. Section 4.2 is on variance checking. Then I define ground signatures and coalgebraic class signatures. Section 4.5 explains the higher-order logic of CCSL and Section 4.6 defines abstract data type specifications. Section 4.7 discusses iterated specifications. The following sections are less formal, Section 4.8 discusses the object-oriented features of CCSL and Section 4.9 combines a few minor topics that do not fit somewhere else. Section 4.10 presents applications of CCSL. In the last section of this chapter I summarise and discuss related work. Most of the sections contain a subsection that describes the concrete CCSL syntax. For convenience the complete CCSL syntax is indexed in the subject index and collected in Appendix B.

4.1. The Type Theory of CCSL

In CCSL class specifications (and abstract data types) may depend on type parameters. For instance, a CCSL specification for the sequences from Section 2.6 depends on one type parameter, the type of the elements of the sequences. When using the sequences in subsequent specifications this type parameter must be instantiated with a concrete type. For CCSL it is therefore necessary to develop a polymorphic type theory. A polymorphic type theory allows one to model parametric polymorphism in the sense of (Cardelli and Wegner, 1985). In such a type theory terms may depend on a finite set of type variables *and* a finite set of term variables. *(Term) judgements* are used to formally derive valid typings for terms. A term judgement consists of four parts, a type variable context, a term variable context, a term, and a type. A typical example is

$$\alpha : \mathsf{Type} \;\mid\; q : \mathsf{Seq}[\alpha] \;\vdash\; \mathsf{next}(q) : \mathbf{1} + \alpha \times \mathsf{Seq}[\alpha]$$

It states that, if the variable q has type $\mathsf{Seq}[\alpha]$ for an arbitrary type α, then the application $\mathsf{next}(q)$ has the depicted type. In the derivation system of a type theory one also has other kinds of judgements, for instance for deriving that a type expression is a valid type in the system. I introduce these different judgements when they are needed.

In the following I present the type theory for CCSL. It is a specialised version of the polymorphic type theory $\lambda\!\rightarrow$ (see Section 3 in (Barendregt, 1992) or Section 8.1 in (Jacobs, 1999a)) enriched with product, coproduct, and exponent types and with the special types Self and Prop. The type Self represents the state space of classes and abstract data types. In Section 4.5 I describe a higher-order logic over the type theory of CCSL. The type Prop will then be the type of propositions. Instead of working with higher-order signatures as (Jacobs, 1999a) does, I prefer to formalise Prop as a special type.

In the type theory of CCSL *type constructors* will play an important role. A type constructor can be thought of as a function that acts on types. A typical example is the type constructor list that builds the type $\mathsf{list}[\tau]$ of (finite) lists over τ for any type τ. The number of arguments that a type constructor takes is called the *arity* of the type constructor. Type constructors of arity 0 (i.e., those that take no arguments) are *type constants* such as $\mathbb{N}$ for the natural numbers. Type constructors are the technical means to extend the specification environment. The technical details about this are in Section 4.7, but it is good to have a rough idea about what is going on.

Each CCSL specification consists of a finite list $\mathcal{S}_1, \ldots, \mathcal{S}_n$ of ground signature extensions (Section 4.3), coalgebraic class specifications (Sections 4.4 and 4.5), and abstract data type specifications (Section 4.6). Each of the $\mathcal{S}_i$ is relative to a ground signature Ω_i and every $\mathcal{S}_i$ can define type constructors, constants, and functions. The newly defined items are added to Ω_i to yield Ω_{i+1}. This way a specification $\mathcal{S}_i$ can refer to all specifications $\mathcal{S}_j$ with $j < i$.

In what follows type constructors are treated as syntactic entities that come along with a semantics. It might help to think of type constructors as resulting from a data type specification (such as finite lists or binary trees) or a process type specification (such as possibly infinite sequences) that already has been processed by the CCSL compiler.

4.1.1. Kinds

Kinds are used to distinguish types from type constructors and to count the type arguments of the type constructors. So kinds are natural numbers. (Other pure type systems, for instance λ_ω allow more complex kinds and also type variables of complex kinds, compare (Barendregt, 1992) or (Jacobs, 1999a).) In judgements I use outlined lowercase letters like $\Bbbk$ for kinds. Ordinary types have kind 0. To improve readability I write $\sigma : \mathsf{Type}$ instead of $\sigma : 0$. Type constructors that take n arguments will have kind n. *Kind judgements* have the form

$$\vdash \Bbbk : \mathsf{Kind}$$

They state that $\Bbbk$ is a valid kind (i.e., a natural number).

4.1.2. Types

Types are built from type variables and type constructors including product types, coproduct (or sum) types, and exponent types. I use lowercase Greek variables like $\alpha, \beta, \ldots$ to denote type variables. A *type variable context* is a finite list of type variables. I assume that all type variables in a context are distinct. This can formally be ensured by using only type variables $\alpha_1, \alpha_2, \ldots$, but I would like to ignore these technicalities. In the type theory $\lambda{\rightarrow}$ type variables are place holders for types only (and not for arbitrary type constructors). To emphasise this I write type contexts as $\alpha_1 : \mathsf{Type},\ \alpha_2 : \mathsf{Type}, \ldots$ with the explicit kind Type. Arbitrary types are denoted with lower case Greek variables like τ, σ. I use *type judgements* to formally derive all types. A type judgement has the form

$$\Xi \ \vdash \ \tau \ : \ \Bbbk \qquad \text{or} \qquad \Xi \ \vdash \ \tau \ : \ \mathsf{Type}$$

where Ξ is a type variable context, τ is a type expression containing only type variables from Ξ, and $\Bbbk$ is a valid kind. Such a type judgement states that τ is a type constructor of kind $\Bbbk$ (or an ordinary type, if $\Bbbk = \mathsf{Type}$).

Type constructors, which can be used to build composite types, are given as part of a ground signature (see Definition 4.3.1 on page 144 below). A type constructor C of kind (or arity) $\Bbbk$ for a valid kind $\Bbbk$ is given as a judgement

$$\vdash \mathsf{C} : \Bbbk$$

Type constructors of arity 0 will be called type constants. I assume a set $\mathcal{C}$ of type constructors in the following.

Definition 4.1.1 (Types) The types over a set of type constructors $\mathcal{C}$ are finitely generated as the least set including

- α for a type variable $\alpha : \mathsf{Type}$
- K for a type constant $\vdash \mathsf{K} : \mathsf{Type}$ in $\mathcal{C}$
- $\mathsf{C}[\tau_1, \ldots, \tau_\Bbbk]$ for a type constructor $\vdash \mathsf{C} : \Bbbk$ in $\mathcal{C}$ and types $\tau_1, \ldots, \tau_\Bbbk$
- Self, the special type that stands for the carrier set of class specifications and abstract data type specifications
- Prop, the type of propositions
- $\mathbf{1}$, the unit type and $\mathbf{0}$ the empty type
- the product $\tau \times \sigma$, the coproduct $\tau + \sigma$, and the exponent $\tau \Rightarrow \sigma$ for types τ and σ

Figure 4.2 contains a derivation system that allows one to derive a judgement $\Xi \vdash \tau : \mathsf{Type}$ precisely if τ is a type according to the preceding Definition with type variables Ξ.

In the following I assume that the product and the coproduct of types is associative (i.e., $(\tau_1 \times \tau_2) \times \tau_3 \cong \tau_1 \times (\tau_2 \times \tau_3)$ and $(\tau_1 + \tau_2) + \tau_3 \cong \tau_1 + (\tau_2 + \tau_3)$). I assume further the isomorphisms $\mathbf{1} \Rightarrow \tau \;\cong\; \tau$ and $\tau \times \mathbf{1} \;\cong\; \tau$. The semantics of types (Definition 4.2.5 on page 139) will ensure that the corresponding interpretations are isomorphic collections of sets. The exponent $\Rightarrow$ is assumed to associate to the right, that is $\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3 \;=\; \tau_1 \Rightarrow (\tau_2 \Rightarrow \tau_3)$. I omit parenthesis in the following when they do not contribute to readability.

kinds

$$\frac{}{\vdash \mathsf{Type} : \mathsf{Kind}} \qquad \frac{\vdash \mathbb{k} : \mathsf{Kind}}{\vdash \mathbb{k} + 1 : \mathsf{Kind}}$$

type variable **Self** **Prop**

$$\frac{}{\Xi \vdash \alpha : \mathsf{Type}}\ \alpha \in \Xi \qquad \frac{}{\Xi \vdash \mathsf{Self} : \mathsf{Type}} \qquad \frac{}{\Xi \vdash \mathsf{Prop} : \mathsf{Type}}$$

type constructor

$$\frac{\vdash \mathbb{k} : \mathsf{Kind} \quad \Xi \vdash \sigma_1 : \mathsf{Type} \quad \cdots \quad \Xi \vdash \sigma_{\mathbb{k}} : \mathsf{Type}}{\Xi \vdash \mathsf{C}[\sigma_1, \ldots, \sigma_{\mathbb{k}}] : \mathsf{Type}} \quad \text{for } \mathsf{C} : \mathbb{k} \text{ in } \mathcal{C}$$

product type

$$\frac{}{\Xi \vdash \mathbf{1} : \mathsf{Type}} \qquad \frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau \times \sigma : \mathsf{Type}}$$

coproduct type

$$\frac{}{\Xi \vdash \mathbf{0} : \mathsf{Type}} \qquad \frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau + \sigma : \mathsf{Type}}$$

exponent type

$$\frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau \Rightarrow \sigma : \mathsf{Type}}$$

The following two rules are not necessary to build all possible types, but sometimes it is convenient to use them. They can be derived by induction on the structure of derivations.

type context weakening **type substitution**

$$\frac{\Xi \vdash \tau : \mathsf{Type}}{\Xi, \alpha : \mathsf{Type} \vdash \tau : \mathsf{Type}}\ \alpha \notin \Xi \qquad \frac{\Xi, \alpha : \mathsf{Type} \vdash \tau : \mathsf{Type} \quad \Xi \vdash \sigma : \mathsf{Type}}{\Xi \vdash \tau[\sigma/\alpha] : \mathsf{Type}}$$

Figure 4.2.: Derivation system for well-formed types over a set of type constructors $\mathcal{C}$.

4.1.3. Type Expressions in CCSL

CCSL allows all types according to Definition 4.1.1. However there are the following points to note.

- Kinds do not appear in the concrete syntax of CCSL, the CCSL type checker ensures that all type constructors get the right number of arguments.
- Type variables appear as normal identifiers, which have been declared as type parameters.
- There are the two keywords `CARRIER` and `SELF` that represent the special type Self. The keyword `CARRIER` represents Self in abstract data type specifications (see Section 4.6), inside class specifications (Section 4.4) one has to use `SELF`.
- CCSL allows n-ary product types $\sigma_1 \times \cdots \times \sigma_n$. Further, the product of types $\sigma_1 \times \cdots \times \sigma_n$ is written with brackets $\texttt{[}\sigma_1, \ldots, \sigma_n\texttt{]}$ like in PVS.
- The binary coproduct and the unit type are formalised as abstract data types, which are defined in the CCSL prelude (see Section 4.9.8). So there is no concrete syntax for unit and coproduct.
- The empty type is not available for the ISABELLE back end of CCSL. For the PVS back end the empty type is declared in the prelude.
- The type Prop is called bool, it is built-in into the CCSL compiler.

The grammar of CCSL is given in a BNF–like notation. Brackets [...] denote optional components, braces $\{\!|\ldots|\!\}$ denote arbitrary repetition (including zero times), and parenthesis (...) denote grouping. Terminals are set in `UPPERCASE TYPEWRITER`, non–terminals in *lowercase slanted*. The terminal symbols for parenthesis and brackets are written as $\underline{(}$, $\underline{)}$, $\underline{[}$, and $\underline{]}$. For convenience all keywords and nonterminals of the CCSL grammar can be found in the subject index.

The concrete syntax for type expressions in CCSL is given by the following BNF rules.

$$
\begin{array}{lcl}
\mathit{type} & ::= & \texttt{SELF} \\
 & | & \texttt{CARRIER} \\
 & | & \texttt{BOOL} \\
 & | & \underline{[}\ \mathit{type}\ \{\!|\ \texttt{,}\ \mathit{type}\ |\!\}\ \texttt{->}\ \mathit{type}\ \underline{]} \\
 & | & \underline{[}\ \mathit{type}\ \{\!|\ \texttt{,}\ \mathit{type}\ |\!\}\ \underline{]} \\
 & | & \mathit{qualifiedid} \\
 & | & \mathit{identifier}\ \mathit{argumentlist} \\
\mathit{argumentlist} & ::= & \underline{[}\ \mathit{type}\ \{\!|\ \texttt{,}\ \mathit{type}\ |\!\}\ \underline{]}
\end{array}
$$

The form `[`σ_1`, ... ,`σ_n` -> `τ`]` is shorthand for `[[`σ_1`, ... ,`σ_n`] -> `τ`]`. Qualified identifiers are explained in Subsection 4.9.6 below (on page 222). In type expressions a qualified identifier can be either a simple *identifier* (referring to a type variable or a type constant), or an instantiated ground signature name with a type *identifier* declared in that ground signature.

4.2. Variance Checking

This section describes the algorithm that computes variances for type variables and for Self. I follow the ideas from (Schroeder, 1997) but generalise Schroeder's variances such that I get information about the deepest nesting level at which a type variable (or Self) occurs positively and negatively.

Variances are mainly important for the CCSL typechecker. To get a semantics, the type expressions from a signature are translated to functors. Depending on the variance of Self one gets a polynomial functor, an extended polynomial functor, or a higher-order polynomial functor (see Proposition 4.2.8 on page 143). The class of models of the signature has very different properties depending on the functor (compare Chapter 3). Another important point is that in abstract data-type definitions only type expressions with a certain variance are allowed (see Definition 4.6.1 on page 187). The reason for this restriction is that there is no initial semantics for arbitrary signatures (Gunter, 1992). Finally, as a minor point, the CCSL compiler generates simpler output in the common case that a type variable does not occur with mixed variance (compare Proposition 4.2.6 and Subsection 4.2.4 on page 140ff).

In the following I first try to explain variances and the algorithm to compute them on an intuitive level. A precise definition follows in the first subsection, the development there is a bit technical, but there is nothing deep behind. To put it a bit sloppy, the algorithm to compute variances only counts parenthesis.

Informally speaking the *variance* of a type variable (or of Self) tells us if the type variable occurs on the left hand side of $\Rightarrow$, on its right hand side, or on both sides. Let me anticipate some bits of Definition 4.2.5 (interpretation of types) to explain this problem in greater detail. Consider type expressions over a set of type constructors that contains only one constant type, so $\mathcal{C} = \{\mathbb{N} : \mathsf{Type}\}$. If we ignore Self and type variables for a moment we can assign to every type τ a set $[\![\tau]\!]$ as its interpretation. We use $\mathbb{N}$, the set of natural numbers as interpretation of the type constant $\mathbb{N}$ and set $[\![\tau \odot \sigma]\!] = [\![\tau]\!] \odot [\![\sigma]\!]$ for $\odot \in \{\times, +, \Rightarrow\}$. (Here $\times, +$, and $\Rightarrow$ denotes the bicartesian closed structure on **Set**, that is, the cartesian product, the disjoint union, and the function space, respectively; see Example 2.2.1 on page 19.)

Assume now that τ contains a type variable: $\alpha : \mathsf{Type} \vdash \tau : \mathsf{Type}$. Then its interpretation is a mapping that assigns to each set U, which we choose as an interpretation for α, the set $[\![\tau]\!]_U$. If for instance $\tau_1 = \alpha \times \mathbb{N}$ then $[\![\tau_1]\!]_U = U \times \mathbb{N}$. So the semantics of a type that contains type variables is an (set–) indexed collection of sets, where the indices are the interpretation of the type variables. Consider now two sets U and V as possible interpretations of α. A function $h : U \longrightarrow V$ gives in a canonical way rise to a function $[\![\tau_1]\!]_U \longrightarrow [\![\tau_1]\!]_V$. For τ_1 this function is given by $\lambda x : U \times \mathbb{N} \,.\, (h(\pi_1\, x), \pi_2\, x)$. If, for a second example, $\tau_2 = \mathbb{N} \Rightarrow \alpha$ then $[\![\tau_2]\!]_U = \mathbb{N} \Rightarrow U = \{f \mid f : \mathbb{N} \longrightarrow U\}$ and the function $[\![\tau_2]\!]_U \longrightarrow [\![\tau_2]\!]_V$ is given by $\lambda f : \mathbb{N} \longrightarrow U \,.\, h \circ f$.

Complications start if the type variable α occurs on the left hand side of $\Rightarrow$: Consider the type $\tau_3 = \alpha \Rightarrow \mathbb{N}$. This time a function $h : U \longrightarrow V$ induces a function $[\![\tau_3]\!]_V \longrightarrow [\![\tau_3]\!]_U$ *in the opposite direction!* It is given by $\lambda f : V \longrightarrow \mathbb{N} \,.\, f \circ h$. In this case, where the induced function goes into the opposite direction, one says that the type variable α occurs in τ with *negative variance* or alternatively one says that α occurs in τ at a *negative position*. A type variable occurs with *positive variance* (at a *positive position*) if the induced function keeps the direction, as in the preceding paragraph. If the type variable occurs with negative variance within a type expression that occurs itself at a negative position, then the type variable has positive variance again. Consider $\tau_4 = (\alpha \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$. As interpretation we have

$$[\![\tau_4]\!]_U \quad = \quad \{f \mid f \text{ is a function that maps functions } U \longrightarrow \mathbb{N} \text{ to elements of } \mathbb{N}\}$$

With a function $h : U \longrightarrow V$ we can build the function

$$\lambda f : (U \Rightarrow \mathbb{N}) \longrightarrow \mathbb{N} \;.\; \big(\lambda g : V \longrightarrow \mathbb{N} \,.\, f(g \circ h)\big) \quad : \quad [\![\tau_4]\!]_U \longrightarrow [\![\tau_4]\!]_V$$

To distinguish the types τ_1 and τ_2 from τ_4 one says that α occurs in τ_1 and τ_2 *strictly positively.*

A type variable can also occur several times (with different variances) in one type expression. Consider $\tau_5 = \alpha \Rightarrow (\alpha \times \mathbb{N})$. To get a function $[\![\tau_5]\!]_U \longrightarrow [\![\tau_5]\!]_V$ we need now *two functions* $h^+ : U \longrightarrow V$ and $h^- : V \longrightarrow U$. The induced function on the interpretation of τ is

$$\lambda f : U \longrightarrow U \times \mathbb{N} \;.\; \big[\lambda v : V \,.\, \big(\, h^+(\pi_1(f(h^-\, v))),\; \pi_2(f(h^-\, v)) \,\big)\big]$$

A type variable that occurs with both positive and negative variance is said to have *mixed variance.*

In general the semantics of types is given by functors. The special type Self serves as a place holder for the arguments of the functor. So for the type $\sigma_1 = \mathsf{Self} \times \mathbb{N}$ we get as semantics the functor $T_1(X) = X \times \mathbb{N}$. And for $\sigma_2 = (\mathsf{Self} \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$ we get the functor $T_2(X) = (X \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$. In both types σ_1 and σ_2 the type Self occurs with positive variance. However T_1 is a polynomial functor whereas T_2 is a higher-order polynomial functor. Because polynomial functors and higher-order polynomial functors have different properties, the CCSL compiler must generate different output, depending on whether a signature corresponds to a polynomial functor or a higher-order polynomial functor. So it is not only important if a type variable (or Self) occurs positively or negatively, it is also important at which maximum nesting level a type variable occurs. Therefore I use pairs (u_-, u_+) of natural numbers to denote variances. The first component u_- denotes the maximum nesting level at which the type variable occurs at a negative position. The second component u_+ denotes the maximum nesting level for positive occurrences. In the next subsection I formalise these variances as a special algebra and give an algorithm that computes the variances of the type variables and of Self in a type expression. In the following I describe informally how variances can be computed.

To compute the variances of a type expression τ one has to annotate every subexpression of τ with a natural number in the following way: The whole type is annotated with 0, walk now recursively down the structure of τ and increase the current number every time you pass over to the left hand side of a $\Rightarrow$. Keep the number constant if you pass over $\times$ or $+$ or if you stay on the right hand side of $\Rightarrow$. Let me write the annotations as subscripts to parenthesis, which enclose the subexpressions. Then I get for τ_4

$$(((\alpha)_2 \Rightarrow (\mathbb{N})_1)_1 \Rightarrow (\mathbb{N})_0)_0$$

Observe that subexpressions at positive positions get even numbers and subexpression at negative positions get odd numbers. This is because the variance is toggled from positive to negative and vice versa on the left hand side of $\Rightarrow$. To get the variance for the type variable α, pick out the maximum annotation of α for all negative occurrences (i.e., the maximal odd number for α) and the maximum annotation for all positive occurrences of α (i.e., the maximal even number). Use ? if the type variable does not occur with the corresponding variance. So we get that α has variance $(?, 2)$ in τ_4. For a more complex example consider the type expression

$$\Big[\big[\big((\alpha)_3 \Rightarrow (\mathbb{N})_2\big)_2 \Rightarrow (\alpha)_1\big]_1 \Rightarrow \big[\big((\alpha)_2 \Rightarrow (\mathbb{N})_1\big)_1 \Rightarrow (\alpha)_0\big]_0\Big]_0$$

Here α has the variance $(3, 2)$.

For the computation of variances also nonconstant type constructors are important. For the discussion of variances one can think of a type constructor as a macro that can be expanded into a type expression. For example let

$$\mathsf{C}[\alpha, \beta] \quad \stackrel{\text{def}}{=} \quad ((\alpha \Rightarrow \beta) \Rightarrow \mathbb{N}) \Rightarrow \mathbb{N}$$

Now we can form the type $\tau_5 = \mathsf{C}[\mathsf{Self}, \alpha] \Rightarrow \mathbb{N}$. After expanding C we can compute the variances and get that Self occurs in τ_5 with variance $(?, 4)$ and α with $(3, ?)$. So for the first argument of C the variance is toggled and the nesting level is increased by three. To denote this I say that the first argument of C has variance $(3, ?)$. Similarly, the second argument of C has variance $(?, 2)$. A type constructor can also copy its arguments into a positive and a negative position, for instance:

$$\mathsf{C}'[\alpha] \quad \overset{\text{def}}{=} \quad \alpha \Rightarrow \alpha$$

Therefore the variance of the only argument of C' is $(1, 0)$

For the formal treatment of type constructors I assume that, from now on, type constructors are given with variance annotation by a sequent

$$\vdash \mathsf{C} :: [(3, ?); (?, 2)]$$

Formally the variance annotation is a finite list of variances. The length of the list equals the arity of the type constructor and the elements of the list stand for the variances of its arguments.

The preceding algorithm to compute variances works only well if all variances of the type constructor are either of form $(?, u_+)$ or of form $(u_-, ?)$. For such a type constructor one adds either u_+ or u_- to the current number. So in the type expression

$$(\mathsf{C}[(\alpha)_4, (\alpha)_3])_1 \Rightarrow (\mathbb{N})_0)_0$$

α has variance $(3, 4)$. The treatment of arbitrary variances for type constructors does not really fit in this simplified annotation algorithm. For arbitrary type expressions it is best to use the algorithm from the next subsection.

I prefer to formalise the product, the coproduct, and the exponent of types and the type Prop by giving extra rules for them. Alternatively one can consider them as type constructors with the following variance:

$$\begin{array}{lcl} \vdash & \mathsf{Prop} & :: [] \\ \vdash & \mathbf{1} & :: [] \\ \vdash & \mathbf{0} & :: [] \\ \vdash & \times & :: [(?, 0); (?, 0)] \\ \vdash & + & :: [(?, 0); (?, 0)] \\ \vdash & \Rightarrow & :: [(1, ?); (?, 0)] \end{array}$$

4.2.1. Formalising Variances

To formalise variance checking I need the natural numbers enriched with an additional element ?, which is a zero for addition.[2] So set $\mathbb{N}^? \overset{\text{def}}{=} \mathbb{N} \cup \{?\}$ and extend addition to $\mathbb{N}^?$ with $? + n = n + ? = ?$ for all $n \in \mathbb{N}^?$. I further extend the order $<$ and make ? the least element: $\forall i \in \mathbb{N} \,.\, ? < i$. Now I can use max with the extended natural numbers, for instance $\mathsf{max}(?, n) = \mathsf{max}(n, ?) = n$ for all $n \in \mathbb{N}^?$. Although we saw in the last section that variances are pairs of an odd and an even number it is technically easier to define them as pairs of $\mathbb{N}^?$.

[2] Recall that the number zero is a one (i.e., a neutral element) for addition.

Definition 4.2.1 The *variance algebra* is the triple $(V, \cdot, \vee)$ such that

- $V = \mathbb{N}^? \times \mathbb{N}^?$ is the set of variances, where ? abbreviates $(?, ?) \in V$,
- $\cdot : V \times V \longrightarrow V$ is the *substitution operation* defined by

$$(u_-, u_+) \;\cdot\; (v_-, v_+) \quad \overset{\text{def}}{=} \quad \big(\mathsf{max}(u_- + v_+, u_+ + v_-),\ \mathsf{max}(u_- + v_-, u_+ + v_+)\big)$$

- and $\vee : V \times V \longrightarrow V$ is the *join operation* given by

$$(u_-, u_+) \;\vee\; (v_-, v_+) \quad \overset{\text{def}}{=} \quad \big(\mathsf{max}(u_-, v_-),\ \mathsf{max}(u_+, v_+)\big)$$

The set of well-formed variances $\overline{V} \subseteq V$ is given by

$$\overline{V} \quad \overset{\text{def}}{=} \quad \{(u_-, u_+) \mid (u_- = ? \text{ or } u_- \text{ is odd}) \wedge (u_+ = ? \text{ or } u_+ \text{ is even})\}$$

In the following I assume that $\cdot$ binds tighter than $\vee$, so $u_1 \cdot u_2 \vee u_3 = (u_1 \cdot u_2) \vee u_3$.

Lemma 4.2.2

1. *$(\overline{V}, \cdot, \vee)$ is a subalgebra of the variance algebra, that is, both the substitution and the join operation restrict to $\overline{V} \times \overline{V} \longrightarrow \overline{V}$.*
2. *$(V, \cdot)$ forms a commutative monoid with zero element ?.*
3. *$(V, \vee)$ forms a commutative monoid with identity ?.*
4. *The substitution operation $\cdot$ distributes over $\vee$, that is $u \cdot (v \vee w) = (u \cdot v) \vee (u \cdot w)$*

Proof The proofs are straightforward computations, they have all been formalised in PVS. □

The preceding definition of variances generalises the variances in (Schroeder, 1997). There, Schroeder uses the finite set $\{+, -, *, ?\}$ as variances, denoting positive, negative, mixed, and unknown variance, respectively. The substitution and join operations are given by

$\cdot$	?	+	−	*
?	?	?	?	?
+	?	+	−	*
−	?	−	+	*
*	?	*	*	*

$\vee$	?	+	−	*
?	?	+	−	*
+	+	+	*	*
−	−	*	−	*
*	*	*	*	*

There is an algebra morphism from my variances to Schroeder's variances. It sends $(?, ?)$ to ?, $(?, u_+)$ to +, $(u_-, ?)$ to −, and all other elements of V to $*$.

Next I present the variance checking algorithm. For a type judgement $\Xi \vdash \tau : \mathsf{Type}$ the variance checking algorithm assigns to every type variable $\alpha \in \Xi$ and to Self a well-formed variance from $\overline{V}$. I give the algorithm by annotating type judgements and the derivation system for types from Figure 4.2. Type judgements have now the form

$$\alpha_1 :: v_1, \dots, \alpha_n :: v_n,\ \mathsf{Self} :: v \quad \vdash \quad \tau : \mathsf{Type}$$

for $v, v_1, \dots, v_n \in \overline{V}$. It is the formal statement that τ is a type where the type variables α_i have variance v_i and Self has variance v in τ. Note that in the above judgement Self does not belong to the type variable context. It is just that I did not find a better way to incorporate the variance of Self into judgements. However, the notation is not completely misleading: Instead of making Self a special type (as I preferred in Definition 4.1.1) one could formalise Self as a distinct type variable. Indeed, the variance algorithm that follows treats Self exactly like a type variable.

In the following I assume that the type constructors in $\mathcal{C}$ are given with variance annotations as judgements

$$\vdash \mathsf{C} :: [v_1, \dots, v_{\Bbbk}]$$

where $[v_1, \dots, v_{\Bbbk}]$ is a finite list of length $\Bbbk$ over $\overline{V}$ and $\Bbbk$ is the arity of C. Type constants are given as $\vdash \mathsf{K} :: []$.

Consider the most basic well-formed type $\Xi \vdash \alpha : \mathsf{Type}$. The type variable α occurs with variance $(?, 0)$ and all other type variables from Ξ and Self do not occur (so they have variance $(?, ?)$). So the new rule for type variables is obviously

type variable

$$\frac{}{\alpha_1 :: ?, \dots, \alpha_{i-1} :: ?,\ \alpha_i :: (?, 0),\ \alpha_{i+1} :: ?, \dots, \alpha_n :: ?,\ \mathsf{Self} :: ? \ \vdash\ \alpha_i : \mathsf{Type}}$$

The rule for type constructors is the most difficult one, so let me postpone it for a moment. The other obvious rules are those for Self, $\mathbf{1}$, and $\mathbf{0}$. In the following rules I abbreviate an arbitrary type variable context $\alpha_1 :: u_1, \dots, \alpha_n :: u_n$ by writing $\alpha_i :: u_i$.

Self

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: (?, 0)\ \vdash\ \mathsf{Self} : \mathsf{Type}}$$

unit type

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: ?\ \vdash\ \mathbf{1} : \mathsf{Type}}$$

empty type

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: ?\ \vdash\ \mathbf{0} : \mathsf{Type}}$$

Prop

$$\frac{}{\alpha_i :: ?,\ \mathsf{Self} :: ?\ \vdash\ \mathsf{Prop} : \mathsf{Type}}$$

The product and the coproduct do not change the variances. We only have to keep in mind, that in $\sigma \times \tau$ every type variable might occur in both types σ and τ, so we have to join the variances.

product

$$\frac{\alpha_i :: u_i,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type} \qquad \alpha_i :: v_i,\ \mathsf{Self} :: v\ \vdash\ \sigma : \mathsf{Type}}{\alpha_i :: (u_i \vee v_i),\ \mathsf{Self} :: (u \vee v)\ \vdash\ \tau \times \sigma : \mathsf{Type}}$$

coproduct

$$\frac{\alpha_i :: u_i,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type} \qquad \alpha_i :: v_i,\ \mathsf{Self} :: v\ \vdash\ \sigma : \mathsf{Type}}{\alpha_i :: (u_i \vee v_i),\ \mathsf{Self} :: (u \vee v)\ \vdash\ \tau + \sigma : \mathsf{Type}}$$

For the exponent $\sigma \Rightarrow \tau$ we also have to join the variances for each type variable. Thereby we have to keep in mind that for all type variables in σ the variances flip around and the nesting level is increased by one. Formally this is done by applying the substitution operation with $(1, ?)$. Note that $(1, ?) \cdot (u_-, u_+) = (u_+ + 1, u_- + 1)$.

exponent type

$$\frac{\alpha_i :: u_i,\ \mathsf{Self} :: u\ \vdash\ \sigma : \mathsf{Type} \qquad \alpha_i :: v_i,\ \mathsf{Self} :: v\ \vdash\ \tau : \mathsf{Type}}{\alpha_i :: ((1, ?) \cdot u_i \vee v_i),\ \mathsf{Self} :: ((1, ?) \cdot u \vee v)\ \vdash\ \sigma \Rightarrow \tau : \mathsf{Type}}$$

The rule for the type constructor is a generalised version of the rule for the exponent. Assume a type constructor $\mathsf{C} :: [u_1, \ldots, u_{\mathbb{k}}]$ in $\mathcal{C}$. Before joining the variances from all the types σ_j we have to apply the substitution operation with the variances u_j.

type constructor

$$\frac{\alpha_i :: v_i^1,\ \mathsf{Self} :: v^1 \vdash \sigma_1 : \mathsf{Type} \quad \cdots \quad \alpha_i :: v_i^{\mathbb{k}},\ \mathsf{Self} :: v^{\mathbb{k}} \vdash \sigma_{\mathbb{k}} : \mathsf{Type}}{\alpha_i :: \overline{v_i},\ \mathsf{Self} :: \overline{v}\ \vdash\ \mathsf{C}[\sigma_1, \ldots, \sigma_{\mathbb{k}}] : \mathsf{Type}}$$

where
$$\begin{array}{rcl} \overline{v_i} & = & u_1 \cdot v_i^1 \ \vee\ \cdots\ \vee\ u_{\mathbb{k}} \cdot v_i^{\mathbb{k}} \\ \overline{v} & = & u_1 \cdot v^1 \ \vee\ \cdots\ \vee\ u_{\mathbb{k}} \cdot v^{\mathbb{k}} \end{array}$$

For completeness I also show the rules for weakening and substitution. Both rules are derivable.

type context weakening

$$\frac{\alpha_1 :: u_1, \ldots, \alpha_n :: u_n,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type}}{\alpha_1 :: u_1, \ldots, \alpha_n :: u_n,\ \alpha_{n+1} :: ?,\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type}}$$

type substitution

$$\frac{\begin{array}{r} \alpha_1 :: u_1, \ldots, \alpha_n :: u_n, \alpha_{n+1} :: u_{n+1},\ \mathsf{Self} :: u\ \vdash\ \tau : \mathsf{Type} \\ \alpha_1 :: v_1, \ldots, \alpha_n :: v_n,\ \mathsf{Self} :: v\ \vdash\ \sigma : \mathsf{Type} \end{array}}{\alpha_1 :: \overline{u_1}, \ldots, \alpha_n :: \overline{u_n},\ \mathsf{Self} :: \overline{u}\ \vdash\ \tau[\sigma/\alpha_{n+1}] : \mathsf{Type}}$$

where
$$\begin{array}{rcl} \overline{u_i} & = & (u_{n+1} \cdot v_i) \vee u_i \\ \overline{u} & = & (u_{n+1} \cdot v) \vee u \end{array}$$

$$\begin{array}{rcll}
\mathcal{V}_x(x) & = & (?,0) & \\
\mathcal{V}_x(\alpha) & = & (?,?) & \text{for } x \neq \alpha \\
\mathcal{V}_x(\mathbf{0}) & = & (?,?) & \\
\mathcal{V}_x(\mathbf{1}) & = & (?,?) & \\
\mathcal{V}_x(\mathsf{Prop}) & = & (?,?) & \\
\mathcal{V}_x(\mathsf{Self}) & = & (?,?) & \text{for } x \neq \mathsf{Self} \\
\mathcal{V}_x(\sigma_1 \times \sigma_2) & = & \mathcal{V}_x(\sigma_1) \vee \mathcal{V}_x(\sigma_2) & \\
\mathcal{V}_x(\sigma_1 + \sigma_2) & = & \mathcal{V}_x(\sigma_1) \vee \mathcal{V}_x(\sigma_2) & \\
\mathcal{V}_x(\sigma_1 \Rightarrow \sigma_2) & = & (1,?) \cdot \mathcal{V}_x(\sigma_1) \ \vee \ \mathcal{V}_x(\sigma_2) & \\
\mathcal{V}_x(\mathsf{C}[\sigma_1,\ldots,\sigma_{\Bbbk}]) & = & u_1 \cdot \mathcal{V}_x(\sigma_1) \vee \cdots \vee u_{\Bbbk} \cdot \mathcal{V}_x(\sigma_{\Bbbk}) & \text{for } \vdash \mathsf{C} :: [u_1,\ldots,u_{\Bbbk}]
\end{array}$$

Figure 4.3.: A top down algorithm for computing the variance $\mathcal{V}_x(\tau)$ of type τ with respect to x

Now it is possible to give a simple and precise definition for the terms positive, negative and mixed variance.

Definition 4.2.3 Let τ be a type such that $\Gamma, \alpha :: (v_-, v_+),\ \mathsf{Self} :: (u_-, u_+) \vdash \tau : \mathsf{Type}$ is derivable. The type variable α occurs in τ with

- *strictly positive variance* if $v_- = ?$ and $v_+ \leq 0$
- *positive variance* if $v_- = ?$
- *negative variance* if $v_+ = ?$
- *mixed variance* if $v_- \neq ?$ and $v_+ \neq ?$

Similarly for Self and (u_-, u_+).

4.2.2. Variance Checking in CCSL

For CCSL it is more useful to have an algorithm that works top–down (instead of bottom–up like a derivation system). Let x be Self or a type variable. Then $\mathcal{V}_x(\tau)$ denotes the variance of x in τ. It is defined by induction on the structure of τ, see Figure 4.3.

Proposition 4.2.4 *The function $\mathcal{V}_x$ computes the variances as defined by the derivation system. More precisely, let τ be a type that contains the type variables $\alpha_1, \ldots, \alpha_n$. Then one can derive the following judgement:*

$$\alpha_i :: \mathcal{V}_{\alpha_i}(\tau),\ \mathsf{Self} :: \mathcal{V}_{\mathsf{Self}}(\tau) \ \vdash \ \tau : \mathsf{Type}$$

Further $\mathcal{V}_\beta(\tau) = (?,?)$ if $\beta \notin \{\alpha_1, \ldots, \alpha_n\}$.

Proof By induction on the structure of types. The induction steps are immediate. □

The function $\mathcal{V}_x$ can be further optimised into a tail-recursive function by storing the variance of the type variable or of Self in a reference cell. This reference cell will be initialised with ?, the neutral element for $\vee$. Then the equations from Figure 4.3 can be transformed into tail recursive form because $\cdot$ distributes over $\vee$ (Lemma 4.2.2 (4)). This tail-recursive variant of the equations in Figure 4.3 is used in the CCSL-compiler.

The variances presented here are not sufficient to distinguish extended cartesian functors from extended polynomial functors. The CCSL compiler uses an additional check to determine if a type corresponds to an extended cartesian functor.

Variances appear in CCSL specifications as annotations to type parameters. They restrict the variance of the annotated type parameter. Their concrete syntax is as follows.

$$\begin{array}{lcl} \textit{variance} & ::= & \texttt{POS} \\ & | & \texttt{NEG} \\ & | & \texttt{MIXED} \\ & | & \underline{(}\ \textit{numberorquestion}\ \textbf{,}\ \textit{numberorquestion}\ \underline{)} \\ \textit{numberorquestion} & ::= & ? \\ & | & \textit{number} \end{array}$$

In addition to the variances of Definition 4.2.1 the CCSL compiler recognises the keywords POS, NEG and MIXED as variances. These latter three variances denote an infinite nesting level and must be used for type constructors of ground signatures. The compiler extends the join and substitution of variances in the obvious way, for instance $\texttt{POS} \vee (?, 2) = \texttt{POS}$, $\texttt{POS} \vee (3, ?) = \texttt{MIXED}$, and $\texttt{POS} \cdot (3, ?) = \texttt{NEG}$. Variances given as a pair of numbers (or question marks) must be proper variances.

4.2.3. Semantics of Types

The discussion on negative variances at the beginning of this section showed that for the semantics of types some notions (for instance the map–combinator) differ with respect to the variance of type variables. Here I take the general approach and develop a semantics for types under the assumption that all type variables and Self occur with mixed variance. For type variables that occur only with positive or negative variance considerable simplifications are possible, see the lemmas below and the next subsection. For the semantics of types I thereby deliberately deviate from the CCSL compiler at the advantage of a clearer presentation.

In the standard case every type $\Xi \vdash \tau : \mathsf{Type}$ gives rise to an indexed collection of functors

$$\bigl(\llbracket\tau\rrbracket_{U_1^-,U_1^+,U_2^-,U_2^+,\dots,U_n^-,U_n^+} \;:\; \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}\bigr)_{U_i^-,U_i^+ \in |\mathbf{Set}|}$$

where n is the number of type variables in Ξ. The indices $U_1^-, \dots, U_n^+$ are sets for the interpretation of the type variables. The set U_i^- is used for the negative occurrences of

α_i and U_i^+ for the positive ones. The arguments of the functor itself are used for the negative and positive occurrences of Self, respectively. In the following I abbreviate the list of indices as $U_i^{-/+}$ if there is no danger of confusion.

The standard case, to which I refer in the previous paragraph, is the case where for every type constructor of arity $\Bbbk$ there is an interpretation functor taking $2\Bbbk$ arguments. The arguments are doubled to separate them into positive and negative occurrences. The following definition deals with the standard case only, I discuss some abnormalities below.

Definition 4.2.5 (Interpretation of Types) Let $\mathcal{C}$ be a set of type constructors.

1. Let $\vdash \mathsf{C} :: [v_1, \ldots, v_{\Bbbk}]$ be a type constructor of arity $\Bbbk$ and let $(-)^{\Bbbk}$ denote the $\Bbbk$–fold product. An *interpretation* of C is a functor

$$[\![\mathsf{C}]\!] : (\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set})^{\Bbbk} \longrightarrow \mathbf{Set}$$

if, for arbitrary sets $V, V', U_1, \ldots, U_{2n}$, it has the following property

   - if the i-th argument of C has positive variance then $[\![\mathsf{C}]\!]$ is constant in its $(2i-1)$-th argument (which interprets the negative occurrences of the i-th argument):

$$\begin{aligned}[\![\mathsf{C}]\!](U_1, \ldots, U_{2i-2}, V, U_{2i}, \ldots, U_{2n}) \quad = \\ [\![\mathsf{C}]\!](U_1, \ldots, U_{2i-2}, V', U_{2i}, \ldots, U_{2n})\end{aligned}$$

   - if the i-th argument of C has negative variance then $[\![\mathsf{C}]\!]$ is constant in its $2i$-th argument (interpreting the positive occurrences of the i-th argument).
   - if the i-th argument of C has unknown variance then $[\![\mathsf{C}]\!]$ is constant in both its $(2i-1)$-th and its $2i$-th argument.

2. Let $\alpha_1, \ldots, \alpha_n \vdash \tau : \mathsf{Type}$ be a type with type constructors from $\mathcal{C}$. Assume that for every $\mathsf{C} \in \mathcal{C}$ we have an interpretation $[\![\mathsf{C}]\!]$. The interpretation of τ is defined by induction on the structure of types.

$$\begin{array}{rcllll}
[\![\alpha_i]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & U_i^+ \\
[\![\mathsf{Self}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & X \\
[\![\mathbf{1}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & \mathbf{1} & = & \{*\} \\
[\![\mathbf{0}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & \mathbf{0} & = & \emptyset \\
[\![\mathsf{Prop}]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & \mathsf{bool} & = & \{\bot, \top\} \\
[\![\sigma_1 + \sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & [\![\sigma_1]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & + & [\![\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) \\
[\![\sigma_1 \times \sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & = & [\![\sigma_1]\!]_{U_1^-,\ldots,U_n^+}(Y,X) & \times & [\![\sigma_2]\!]_{U_1^-,\ldots,U_n^+}(Y,X)
\end{array}$$

$$\begin{aligned}
[\![\sigma_1 \Rightarrow \sigma_2]\!]_{U_1^-,\dots,U_n^+}(Y,X) &= [\![\sigma_1]\!]_{U_1^+,U_1^-,U_2^+,U_2^-,\dots,U_n^+,U_n^-}(X,Y) \quad \Rightarrow \\
&\qquad [\![\sigma_2]\!]_{U_1^-,U_1^+,U_2^-,U_2^+,\dots,U_n^-,U_n^+}(Y,X) \\
[\![\mathsf{C}[\sigma_1,\dots,\sigma_k]]\!]_{U_1^-,\dots,U_n^+}(Y,X) &= [\![\mathsf{C}]\!]\Big([\![\sigma_1]\!]_{U_1^+,U_1^-,U_2^+,U_2^-,\dots,U_n^+,U_n^-}(X,Y), \\
&\qquad [\![\sigma_1]\!]_{U_1^-,U_1^+,U_2^-,U_2^+,\dots,U_n^-,U_n^+}(Y,X), \\
&\qquad \vdots \\
&\qquad [\![\sigma_n]\!]_{U_1^+,U_1^-,U_2^+,U_2^-,\dots,U_n^+,U_n^-}(X,Y), \\
&\qquad [\![\sigma_n]\!]_{U_1^-,U_1^+,U_2^-,U_2^+,\dots,U_n^-,U_n^+}(Y,X)\Big)
\end{aligned}$$

The morphism part is defined in the obvious way (by replacing Y and X with suitable functions f^- and f^+).

Observe how the indices and the arguments for positive and negative occurrences are flipped around on the left hand side of the exponent and in the arguments for the functor $[\![\mathsf{C}]\!]$.

The interpretation of a type τ containing n type variables can be extended (in the obvious way) to a functor taking $2n+2$ arguments. Its morphism part is denoted with $[\![\tau]\!]_{\overline{g}}(f^-, f^+)$ for a suitable list of functions $\overline{g} = g_1^-, g_1^+, \dots, g_n^-, g_n^+$.

Under certain circumstances (occurring in conjunction with iterated specifications, see Section 4.7) for some type constructor C only the object mapping of an interpretation functor might be available (i.e., there is no morphism part for $[\![\mathsf{C}]\!]$). In this case the interpretation $[\![\tau]\!]$ degrades to an indexed collection of mappings $|\mathbf{Set}| \times |\mathbf{Set}| \longrightarrow |\mathbf{Set}|$.

In even more obscure cases the interpretation $[\![\mathsf{C}]\!](\cdots U_i^-, U_i^+ \cdots)$ of a type constructor is only defined if the respective arguments for positive and negative occurrences are equal, that is if $U_i^- = U_i^+$ for all i. In this case the interpretation $[\![\tau]\!]_{\dots U_i^-, U_i^+ \dots}(Y,X)$ is only defined for $Y = X$ and $U_i^- = U_i^+$.

If Self occurs in τ with positive variance then the first argument is ignored for every functor in the collection $([\![\tau]\!])$. In this case the interpretation functors factor through $\pi_2 : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \longrightarrow \mathbf{Set}$. Similarly the second argument is ignored if Self occurs in τ with negative variance. Some of the indices $U_i^{-/+}$ are ignored if not all type variables occur with mixed variance in τ.

Proposition 4.2.6 *Let $\alpha_1, \dots, \alpha_n \vdash \tau : \mathsf{Type}$ be a type as before. Let V and V' be arbitrary sets.*

- *If Self occurs in τ with positive variance, then*

$$[\![\tau]\!]_{U_1^-,\dots,U_n^+}(V,X) \quad = \quad [\![\tau]\!]_{U_1^-,\dots,U_n^+}(V',X)$$

- *If Self occurs in τ with negative variance, we have*

$$[\![\tau]\!]_{U_1^-,\dots,U_n^+}(Y,V) \quad = \quad [\![\tau]\!]_{U_1^-,\dots,U_n^+}(Y,V')$$

- *Assume the type variable α_i has positive variance in τ. Then*

$$[\![\tau]\!]_{U_1^-,\dots,V,U_i^+,\dots,U_n^+}(Y,X) \quad = \quad [\![\tau]\!]_{U_1^-,\dots,V',U_i^+,\dots,U_n^+}(Y,X)$$

- *And if α_i has negative variance then*

$$[\![\tau]\!]_{U_1^-,\dots,U_i^-,V,\dots,U_n^+}(Y,X) \quad = \quad [\![\tau]\!]_{U_1^-,\dots,U_i^-,V',\dots,U_n^+}(Y,X)$$

Proof By induction on the structure of types. □

For a type σ that does not contain Self every functor in the interpretation $[\![\sigma]\!]$ is a constant functor (i.e., the result does not depend on the arguments). Therefore I write for the interpretation of such types $[\![\tau]\!]_{U_1^-,\dots,U_n^+}$ instead of $[\![\tau]\!]_{U_1^-,\dots,U_n^+}(Y,X)$. The indexing with the interpretations for the type variables looks complicated but is in fact a rather simple idea. After one got used to this concept the complicated notation distracts from the interesting points. In following sections I will therefore sometimes drop these indexes and simply write $[\![\tau]\!]$ for a fixed interpretation of the type variables and of Self.

4.2.4. Separation of Variances

In this subsection I explain how the CCSL compiler deals with type variables with mixed variance. It is possible to skip this subsection and return later, if questions about this issue remain open.

Ideally the CCSL compiler should implement the semantics of types of Definition 4.2.5 literally. It should do a variance analysis on the input and separate all type variables into their positive and negative occurrences. However, there are the following problems with such a rigorous approach: First, very often there is no type variable with mixed variance in a specification. For this common case Proposition 4.2.6 shows that many of the indices (that interpret a particular variance of a type variable) are superfluous. These superfluous items would probably confuse the users of CCSL. Almost certainly PVS would get confused. The second problem is that it is not possible to separate variances in the semantics of CCSL's logic (described in Section 4.5).

For these reasons the CCSL compiler generally uses only one interpretation for any type variable. If the type variable occurs only positively or only negatively it is otherwise correctly handled according to Definition 4.2.5. For a type expression that contains a type variable α_i with mixed variance the morphism part of the semantics stays undefined. Further, for the object part of the semantics the compiler assumes that the arguments for positive and negative occurrences of α_i are equal, that is that $U_i^+ = U_i^-$. If a type variable with mixed variance poses problems then the compiler issues a warning. For data type specifications and for class specifications without assertions the user can easily separate himself the type variable with mixed variance into a positive and a negative one.

The special type Self is always handled in a correct way (even if it occurs with mixed variance). This ensures that the CCSL compiler generates the correct notion of coalgebra morphism for class signatures as long as for all type constructors the morphism part of their semantics is defined.

Predicate and relation lifting (defined in Subsection 4.4.1 below) are treated differently. For type variables that occur only positively or only negatively the compiler uses one parameter predicate for predicate lifting. So for those type variables there is no separation. However, a type variable that has mixed variance is separated into its positive and its negative positions. For such a type variable the compiler introduces two predicates in the generated code. This guarantees that the definitions for predicate lifting and invariant are correctly generated for all class specifications. Relation lifting is treated analogously. Therefore the generated notions of relation lifting and bisimulation are correctly generated as long as the greatest bisimulation does exists.

4.2.5. Classification of Types

Types are classified according to the variance of Self. Via the interpretation of types there is a correspondence with the classification of functors into polynomial, extended polynomial, and higher-order polynomial functors.

Definition 4.2.7 Let $\mathcal{C}$ be a set of type constructors with variance annotations and let τ be a type over $\mathcal{C}$.

1. τ is a *constant type* if $\mathcal{V}_{\mathsf{Self}}(\tau) = ?$.
2. τ is a *polynomial type* if Self occurs only at strictly covariant positions in τ, that is if $\mathcal{V}_{\mathsf{Self}}(\tau) = (?, u_+)$ for $u_+ \leq 0$.
3. τ is an *extended polynomial type* if $\mathcal{V}_{\mathsf{Self}}(\tau) = (u_-, u_+)$ for $u_- \leq 1$ and $u_+ \leq 0$.
4. τ is a *higher-order polynomial type*, if it is not extended polynomial.
5. τ is a *constructor type* in case $\tau = \sigma \Rightarrow \mathsf{Self}$ and σ is a polynomial type.
6. τ is a *constant constructor type* if $\tau = \sigma \Rightarrow \mathsf{Self}$ is a constructor type and if additionally σ is a constant type.
7. τ is a *method type* if $\tau = (\mathsf{Self} \times \sigma) \Rightarrow \rho$.
8. A method type $\tau = (\mathsf{Self} \times \sigma) \Rightarrow \rho$ is a *polynomial/extended-polynomial/higher-order polynomial method type* if $\sigma \Rightarrow \rho$ is a polynomial/extended-polynomial/higher-order polynomial type.

Note that via the isomorphism $\tau \times \mathbf{1} \cong \tau$ and the associativity of $\times$ also $\mathsf{Self} \Rightarrow \rho$ and $\mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \Rightarrow \rho$ are method types for arbitrary $\sigma, \sigma_1, \ldots, \sigma_n, \rho$. Via the isomorphism $\mathbf{1} \Rightarrow \tau \cong \tau$ the type Self is a constant constructor type.

With the formalisation of variances it is now possible to define the term *binary method* precisely. A method of type $\mathsf{Self} \times \sigma \Rightarrow \rho$ is called a binary method, if Self occurs with negative variance in $\sigma \Rightarrow \rho$. Note that in this case $\mathsf{Self} \times \sigma \Rightarrow \rho$ cannot be a polynomial method type. However not every method of extended-polynomial or higher-order polynomial type is also a binary method. For CCSL the classification of methods into binary and unary methods is not really important. Here only the classification of Definition 4.2.7 is significant.

The use of the attributes polynomial, extended polynomial and higher-order to classify types is justified by the following proposition.

Proposition 4.2.8 *Assume that $\mathcal{C}$ contains only type constants and let $\alpha_1, \ldots, \alpha_n \vdash \tau : \mathsf{Type}$ be an arbitrary type over $\mathcal{C}$. Let $F = [\![\tau]\!]_{U_1^-,\ldots,U_n^+}$ be the interpretation functor for fixed sets $U_1^-, \ldots, U_n^+$. The type τ is a*

$$\left\{\begin{array}{l} \textit{constant} \\ \textit{polynomial} \\ \textit{extended-polynomial} \\ \textit{higher-order} \end{array}\right\} \textit{type precisely if } F \textit{ is a } \left\{\begin{array}{l} \textit{constant} \\ \textit{polynomial} \\ \textit{extended-polynomial} \\ \textit{higher-order} \end{array}\right\} \textit{functor.}$$

Proof By induction on the structure of τ. □

The restriction in the preceding proposition that τ does not contain any nonconstant type constructor is rather severe. The weak requirements for the interpretation of nonconstant type constructors do not allow one to derive anything in general. For the CCSL compiler the situation is slightly better: The compiler can keep track of type constructors that stem from a processed class or data type specification. Types that contain such type constructors give rise to *data functors* in the sense of (Hensel, 1999), but see also (Hensel and Jacobs, 1997; Rößiger, 1999). I discuss this issue in Section 4.7 on iterated specifications.

4.3. Ground Signatures

In this section I define (polymorphic) *ground signatures*. Ground signatures are used to declare types, functions, and constants that are available in the specification environment. For instance one usually expects that the natural numbers $\mathbb{N}$ with addition and multiplication are available. In CCSL ground signatures also serve a second purpose: They make iterated specifications (Section 4.7) possible.

The logic that I define in Section 4.5 places a few restriction on ground signatures and their models. For the semantics of *behavioural equality* and of modal operators every type constructor (of arity greater than zero) in the ground signature must be equipped with two special constants: predicate and relation lifting. This requirement is captured with the notion of *proper ground signatures*.

Definition 4.3.1 (Ground Signature)

- A ground signature Ω consists of
  - a set $|\Omega|$ of type constructors with variance annotations,
  - an indexed set (Ω_σ) of constant symbols for each constant type σ over $|\Omega|$. A constant symbol $f \in \Omega_\sigma$ is given as a (term) judgement $\Xi \mid \emptyset \vdash f : \sigma$ where σ is a constant type such that $\Xi \vdash \sigma : \mathsf{Type}$ is derivable.
- A ground signature is called *plain* if the set $|\Omega|$ contains only type constants (i.e., type constructors of arity zero).
- A ground signature is called *proper* if the set of constant symbols contains at least the following two symbols for every type constructor $\mathsf{C} \in |\Omega|$ of arity $\Bbbk$

$$\begin{aligned}\alpha_1, \ldots, \alpha_\Bbbk \mid \emptyset \vdash \mathsf{Pred}_\mathsf{C} \; : \; & (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow \\ & (\alpha_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}[\alpha_1, \ldots, \alpha_\Bbbk] \Rightarrow \mathsf{Prop}\end{aligned}$$

$$\begin{aligned}\alpha_1, \ldots, \alpha_\Bbbk,\ \beta_1, \ldots \beta_\Bbbk \mid \emptyset \vdash \mathsf{Rel}_\mathsf{C} \; : \; & (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) & \Rightarrow \cdots \Rightarrow (\alpha_\Bbbk \times \beta_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_\Bbbk \times \beta_\Bbbk \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}[\alpha_1, \ldots, \alpha_\Bbbk] \times \mathsf{C}[\beta_1, \ldots, \beta_\Bbbk] \Rightarrow \mathsf{Prop}\end{aligned}$$

  The constant Pred_C is the *predicate lifting* of C and Rel_C is its *relation lifting*. Note that both take $2\Bbbk$ arguments. This is necessary to separate co– and contravariant occurrences.

Example 4.3.2 The CCSL compiler starts with an empty ground signature. Before the user file is read the CCSL *prelude* is processed (see also Subsection 4.9.8). This prelude is a valid CCSL string, which is hard wired into the compiler. So the user file is opened with a (proper) ground signature Ω_P that contains the declarations from the prelude.

The signature Ω_P contains the following type constructors[3]

$$\begin{array}{rcl} \vdash \mathsf{list} & : & [(?, 0)] \\ \vdash \mathsf{Lift} & : & [(?, 0)] \\ \vdash \mathsf{Coproduct} & : & [(?, 0); (?, 0)] \\ \vdash \mathsf{Unit} & : & [] \\ \vdash \mathsf{EmptyType} & : & [] \end{array}$$

The intended semantics (which is ensured by the CCSL compiler in cooperation with the target theorem prover) is that list constructs the finite lists over a given type, $\mathsf{Lift}[\alpha]$ is

[3] The type constructor EmptyType is not defined for the ISABELLE back end.

an abbreviation for $\alpha + \mathbf{1}$ and $\mathsf{Coproduct}[\alpha, \beta] = \alpha + \beta$. The type constructors Unit and EmptyType give the two special types **1** and **0**, respectively. The type constructor Lift is used in CCSL to model partial functions. The coproduct is in the prelude because there is no special syntax for the coproduct of types in CCSL.

Besides the predicate and relation lifting of the three type constructors the ground signature Ω_P contains the following constants:[4]

$$\begin{aligned}
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{null} : \mathsf{list}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{cons} : \alpha \times \mathsf{list}[\alpha] \Rightarrow \mathsf{list}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{null?} : \mathsf{list}[\alpha] \Rightarrow \mathsf{Prop} \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{cons?} : \mathsf{list}[\alpha] \Rightarrow \mathsf{Prop} \\[1ex]
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{bot} : \mathsf{Lift}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{up} : \alpha \Rightarrow \mathsf{Lift}[\alpha] \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{bot?} : \mathsf{Lift}[\alpha] \Rightarrow \mathsf{Prop} \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{up?} : \mathsf{Lift}[\alpha] \Rightarrow \mathsf{Prop} \\[1ex]
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in1} : \alpha \Rightarrow \mathsf{Coproduct}[\alpha, \beta] \\
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in2} : \beta \Rightarrow \mathsf{Coproduct}[\alpha, \beta] \\
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in1?} : \mathsf{Coproduct}[\alpha, \beta] \Rightarrow \mathsf{Prop} \\
\alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{in2?} : \mathsf{Coproduct}[\alpha, \beta] \Rightarrow \mathsf{Prop} \\[1ex]
\emptyset \mid \emptyset &\vdash \mathsf{unit} : \mathsf{Unit} \\[1ex]
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{empty\_fun} : \mathsf{EmptyType} \Rightarrow \alpha
\end{aligned}$$

The constants null, cons, unit, up, in1, in2, and unit are the expected constructors and injections. The other constants are recogniser predicates. The predicate cons?, for instance, is true for a list l if $l = \mathsf{cons}(a, l')$ for some a and l'. Similarly for the other recognisers.

In applications it is nice to have also accessor functions, like $\mathsf{car} : \mathsf{List}[\alpha] \rightharpoonup \alpha$, which delivers the head for nonempty lists. Accessors are usually partial functions (depicted as partial arrow $\rightharpoonup$). The type theory that I developed in this chapter deals (for simplicity) only with total functions. So formally car cannot be incorporated as a constant of type $\mathsf{List}[\alpha] \Rightarrow \alpha$ without leading to inconsistencies. However, the CCSL compiler is a bit more relaxed and declares also the following accessors:

$$\begin{aligned}
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{car} : \mathsf{list}[\alpha] \Rightarrow \alpha \\
\alpha : \mathsf{Type} \mid \emptyset &\vdash \mathsf{cdr} : \mathsf{list}[\alpha] \Rightarrow \mathsf{list}[\alpha]
\end{aligned}$$

[4] In PVS identifiers can contain question marks. The same applies to CCSL. When generating output for ISABELLE the names for the recognisers are is_null, is_cons, and so on. Further for ISABELLE the native ISABELLE constructor names Nil and Cons are used for lists.

$$\alpha : \mathsf{Type} \mid \emptyset \vdash \mathsf{down} : \mathsf{Lift}[\alpha] \Rightarrow \alpha$$

$$\begin{aligned} \alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{out1} : \mathsf{Coproduct}[\alpha,\beta] \Rightarrow \alpha \\ \alpha : \mathsf{Type}, \beta : \mathsf{Type} \mid \emptyset &\vdash \mathsf{out2} : \mathsf{Coproduct}[\alpha,\beta] \Rightarrow \beta \end{aligned}$$

The CCSL type checker treats accessors (erroneously) as total functions. There are no consistency problems for the following reasons: PVS has a type theory with predicate subtypes. There the accessor car is a total function, which has the subtype of nonempty lists as domain. The CCSL compiler uses this correctly typed function as semantics of car. The theorem prover ISABELLE/HOL has no predicate subtypes but its semantics is based in the HOL tradition (Gordon and Melham, 1993) on an universe of nonempty sets. Consequently ISABELLE/HOL does not allow for empty types and there is the special constant arbitrary that inhabits every type.[5] If the CCSL compiler generates output for ISABELLE/HOL then as semantics for car it takes a function that returns arbitrary for the empty list. ■

A *model* of the ground signature contains functors, (polymorphic) functions and constants that can be used to interpret the syntactic symbols in the ground signature. For proper models of proper ground signatures I require two basic properties for the interpretation of predicate and relation lifting. Namely that predicate lifting commutes with truth and that relation lifting commutes with equality in the sense of Lemma 3.3.2.

Definition 4.3.3 (Model of Ground Signature)

- Let Ω be a ground signature. A model of it consists of
  - the object part of an interpretation functor $[\![\mathsf{C}]\!]$ for every type constructor $\mathsf{C} \in |\Omega|$,
  - an indexed family of functions or constants

    $$[\![f]\!]_{U_1,U_2,\ldots U_n} : [\![\sigma]\!]_{U_1,U_1,U_2,U_2,\ldots,U_n,U_n}$$

    for every constant symbol $\alpha_1, \ldots, \alpha_n \mid \emptyset \vdash f : \sigma$ in Ω.
- A model of a proper ground signature is called *proper* if all the interpretations $[\![\mathsf{C}]\!]$ are functors and if additionally the following condition is satisfied: For all type constructors $\mathsf{C} \in |\Omega|$ of arity $\Bbbk$ it holds that

$$\begin{aligned} [\![\mathsf{Pred}_{\mathsf{C}}]\!]_{U_1,\ldots,U_\Bbbk}(\top_{U_1}, \top_{U_1}, \ldots, \top_{U_\Bbbk}, \top_{U_\Bbbk}) &= \top_{[\![\mathsf{C}]\!](U_1,U_1,\ldots,U_\Bbbk,U_\Bbbk)} \\ [\![\mathsf{Rel}_{\mathsf{C}}]\!]_{U_1,\ldots,U_\Bbbk,U_1,\ldots,U_\Bbbk}(\mathrm{Eq}(U_1), \mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_\Bbbk), \mathrm{Eq}(U_\Bbbk)) &= \\ \mathrm{Eq}([\![\mathsf{C}]\!](U_1, U_1, \ldots, U_\Bbbk, U_\Bbbk)) \end{aligned}$$

[5] The constant arbitrary is neither in (Nipkow et al., 2002a) nor in (Nipkow et al., 2002b) described. See the file `src/HOL/HOL.thy` in the ISABELLE distribution.

The two conditions on proper models of ground signatures ensure that also in the presence of type constructors predicate lifting commutes with truth and relation lifting with equality (see Lemma 4.4.9 on page 157 below). In turn this implies that truth is an invariant and equality is a bisimulation (see Proposition 4.4.11 on page 158). On type constants K these two conditions have the following effect: $\mathsf{Pred}_{\mathsf{K}} = \top_{[\![\mathsf{K}]\!]}$ and $\mathsf{Rel}_{\mathsf{K}} = \mathrm{Eq}([\![\mathsf{K}]\!])$. This matches the treatment of constants in predicate and relation lifting for higher-order polynomial functors.

Example 4.3.4 The model for Ω_P maps list to the functor that yields the initial algebra for the functor $F^U_{\mathsf{list}}(X) = X \times U + \mathbf{1}$ for every argument U. For Lift, $\mathsf{Coproduct}$, Unit, $\mathsf{EmptyType}$ and the constant symbols it takes the obvious constructions. The predicate and relation lifting for $\mathsf{Coproduct}$ is given by $+_{\mathrm{P}}$ and $+_{\mathrm{R}}$, respectively. Also the liftings for Lift are obvious: It is $\mathsf{Pred}_{\mathsf{Lift}}(P) = P +_{\mathrm{P}} \top_{\mathbf{1}}$ and $\mathsf{Rel}_{\mathsf{Lift}}(R) = R +_{\mathrm{R}} \mathrm{Eq}(\mathbf{1})$. The liftings for list are defined by induction and follow the general description in Section 4.7 below (I ignore the contravariant arguments):

$$\begin{array}{rcl}
[\![\mathsf{Pred}_{\mathsf{list}}]\!](P)(\mathsf{null}) & = & \top \\
[\![\mathsf{Pred}_{\mathsf{list}}]\!](P)(\mathsf{cons}(a,l)) & = & P(a) \;\wedge\; [\![\mathsf{Pred}_{\mathsf{list}}]\!](P)(l) \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{null},\mathsf{null}) & = & \top \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{cons}(a,l),\mathsf{null}) & = & \bot \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{null},\mathsf{cons}(a,l)) & = & \bot \\
[\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(\mathsf{cons}(a,l),\mathsf{cons}(a',l')) & = & R(a,a') \;\wedge\; [\![\mathsf{Rel}_{\mathsf{list}}]\!](R)(l,l')
\end{array}$$

■

The ground signature describes types and operations that are available in the environment. So usually a model for it is provided (automatically) by the environment. The CCSL compiler for instance assumes that all symbols in the ground signature are defined in the target theorem prover. Therefore it treats symbols from the ground signature literally: their semantics is the same symbol again. In the following sections I assume a proper ground signature Ω and a proper model $\mathcal{M}_\Omega$ of it. Types that appear will be types over $|\Omega|$ and their interpretation will be the interpretation with respect to $\mathcal{M}_\Omega$.

4.3.1. Ground Signatures in CCSL

Ground signatures in CCSL are actually ground signature extensions. As explained before the CCSL compiler keeps a current ground signature while parsing the source file. A ground signature declaration extends this current ground signature with type constructors and constants. These items must be defined either in the ground signature itself or in the target theorem prover. The concrete grammar is as follows.

$$\begin{array}{rcl}
\textit{groundsignature} & ::= & \texttt{BEGIN}\ \textit{identifier}\ [\ \textit{parameterlist}\ ]\ \texttt{:}\ \texttt{GROUNDSIGNATURE} \\
 & & \{\!\!\{\ \textit{importing}\ \}\!\!\}\ \{\!\!\{\ \textit{signaturesection}\ \}\!\!\} \\
 & & \texttt{END}\ \textit{identifier}
\end{array}$$

parameterlist	::=	[*parameters* ⦃ , *parameters* ⦄]
parameters	::=	*identifier* ⦃ , *identifier* ⦄ : [*variance*] `TYPE`

A Ground signature (extension) starts with the keyword `BEGIN`, followed by the name of the ground signature, an optional (global) type parameter list and the keyword `GROUNDSIGNATURE`. The type parameters build a type variable context for all declarations in the ground signature. In CCSL it is necessary to declare type variables as type parameters because there is no special syntax to distinguish type variables from other identifiers.

Any type parameter can get a variance annotation. The variance annotation is compulsory for all type parameters if the ground signature declares a type constructor without giving its definition.

Importing clauses are explained in Subsection 4.9.4. For ground signatures they are necessary, if the items that are declared in the ground signature require extra theories to be loaded in the target theorem prover.

The body of a ground signature contains an arbitrary number of sections, declaring or defining type constructors and (possibly) polymorphic constants or functions.

signaturesection	::=	*typedef*
	|	*signaturesymbolsection* [;]
typedef	::=	`TYPE` *identifier* [*parameterlist*] [`=` *type*]
signaturesymbolsection	::=	`CONSTANT` *termdef* ⦃ ; *termdef* ⦄
termdef	::=	*idorinfix* [*parameterlist*] : *type* [*formula*]
idorinfix	::=	(*infix_operator*)
	|	*identifier*

Each item in a ground signature can declare additional (local) type parameters in a separate parameter list. The type variable context of an item is given by the concatenation of the global parameter list with the local parameter list of that item. The local type parameters are syntactic sugar. They are convenient if only a few items in one ground signature require additional type parameters.

Type constructors are introduced with the keyword `TYPE`. The arity of the new type constructor is defined as the number of the declared (global and local) type parameters. If the optional type expression is present, then the type constructor is defined in CCSL. In this case the CCSL compiler derives the variances annotations, the predicate and relation lifting, and the morphism component of (the semantics of) the type constructor.

If the type constructor is not defined (the optional type expression is left out) then all type parameter must have variance annotations to allow the compiler to derive the variance of the type constructor. For such declarations the compiler assumes that the type constructor and its liftings are defined in the target theorem prover as functions

```
Begin SetSig [ U : Neg Type ] : GroundSignature
  Type set = [U -> bool]
  Constant
    empty : set[U]
    empty = Lambda(t : U) : false;

    intersect : [set[U], set[U] -> set[U]];
    (+)       : [set[U], set[U] -> set[U]];

    ( * ) [V : Type] : [set[U], set[V] -> set[[U,V]]]
    (S * R)(u,v) = ( S u And R v );
End SetSig
```

Figure 4.4.: A rudimentary ground signature extension for power sets

of an appropriate type. These functions are assumed to fulfil the conditions for proper models of ground signatures. Their names are derived from the name of the type constructor by appending the suffixes “Pred”, “Rel”, and “Map”.

Constants and functions are introduced with the keyword CONSTANT. One can also declare infix operators, see Subsection 4.9.5 (on page 221) for the details. The constants can be defined in CCSL by providing a definition in higher-order logic in the syntax formulae (see Subsection 4.5.4). If a formula is present, then it must be an equation and the left hand side of the equation must be the constant to be defined, possibly applied to some variables.

Figure 4.4 contains as example a rudimentary ground signature extension for the (contravariant) powerset type constructor. In addition to the type constructor it declares a constant for the empty set, a function for intersection, the infix operator + for union, and the operator $*$ for the cartesian product. For demonstration purposes I defined some of the constants.

There is also a more lightweight syntax for declaring single types and constants, see Subsection 4.9.7 on anonymous ground signatures (on page 222 below).

4.4. Coalgebraic Class Signatures

This section introduces the structural aspects of coalgebraic specification: signatures and signature models. The following definitions follow very closely what is implemented in the CCSL compiler. The definitions are optimised for practicability — and not for succinctness. Recall from Definition 4.2.7 that method types are types of the form $\mathsf{Self} \times \tau \Rightarrow \tau'$ and constant constructor types have the form $\sigma \Rightarrow \mathsf{Self}$, where Self must not occur in σ.

Definition 4.4.1 (Coalgebraic Class Signature) Assume a set of type constructors $\mathcal{C}$ (possibly stemming from a ground signature). A *coalgebraic class signature* is a pair $\langle \Sigma_M, \Sigma_C \rangle$ where Σ_M is a finite set of method declarations $m_i : \tau_i$, for method types τ_i, and Σ_C is a finite set of constructor declarations $c_i : \sigma_i$ for constant constructor types σ_i. The set of type variables occurring in the τ_i and the σ_i are the *type parameters* of the signature.

Example 4.4.2 (Queue Signature) Consider a *first–in–first–out* (*FIFO*) queue. It supports two operations, one for enqueueing elements (put) and one for removing elements from the head (top). Removing the first element from a queue is a partial operation, which fails if the queue is empty. Therefore the signature Σ_{Queue} contains the following method two declarations

$$\begin{aligned} \mathsf{put} &: \mathsf{Self} \times \alpha \longrightarrow \mathsf{Self} \\ \mathsf{top} &: \mathsf{Self} \longrightarrow \mathsf{Lift}[\alpha \times \mathsf{Self}] \end{aligned}$$

Additionally, there is the constructor declaration

$$\mathsf{new} : \mathsf{Self}$$

For any element x of Self we have either $\mathsf{top}(x) = \mathsf{bot}$ (signalling an empty queue) or $\mathsf{top}(x) = \mathsf{up}(a, x')$, where a is the first element of the queue and x' is the successor state of x with a removed. Instead of the simple constructor new, one could also use a constructor $\mathsf{new\_from\_list} : \mathsf{list}[\alpha] \to \mathsf{Self}$ that takes the elements of a list to initialise the queue.

This example of queues is the running example of this and the following section. The example has been fully worked out in CCSL and PVS. The complete sources are available in the world wide web, see Appendix A. ■

In the following I need the term of a *subsignature*. Subsignatures will be used for inheritance, for the visibility modifiers `PUBLIC` and `PRIVATE`, and for the modal operators. The following definition might be a bit surprising on first sight, because it completely neglects constructor declarations. I motivate this decision in the general discussion about inheritance in Subsection 4.8.1 (on page 210) below.

Definition 4.4.3 (Subsignature) Assume a ground signature Ω and let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature. A class signature $\Sigma' = \langle \Sigma'_M, \Sigma'_C \rangle$ is a *subsignature* of Σ, denoted by $\Sigma' \leq \Sigma$, if $\Sigma'_M \subseteq \Sigma_M$.

Coalgebraic class signatures are classified according to the method types they contain. If a signature contains a higher-order (respectively extended polynomial) method type, then I refer to it as a signature with a higher-order method (with an extended polynomial method, respectively).

To define the semantics of class signatures it is necessary to extract type information from the signature. Assume a coalgebraic class signature Σ in the following. Let $m : \tau \in$

Σ_M be a method declaration, so that τ is a method type. Define the operation $\mathcal{T}_M$ as follows

$$\mathcal{T}_M(m) \quad = \quad \begin{cases} \rho & \text{if} \quad \tau = \ \mathsf{Self} \Rightarrow \rho \\ (\sigma_1 \times \cdots \times \sigma_n) \Rightarrow \rho & \text{if} \quad \tau = \ (\mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n) \Rightarrow \rho \end{cases}$$

And if $c : \tau \in \Sigma_C$ is a constructor declaration (for a constructor type τ) set

$$\mathcal{T}_C(c) \quad = \quad \begin{cases} \sigma & \text{if} \quad \tau = \sigma \Rightarrow \mathsf{Self} \\ \mathbf{1} & \text{if} \quad \tau = \mathsf{Self} \end{cases}$$

Let $m_1, \ldots, m_k$ be the method declarations of Σ. The combined method type of Σ, denoted by τ_Σ, is defined as

$$\tau_\Sigma \quad = \quad \mathcal{T}_M(m_1) \times \cdots \times \mathcal{T}_M(m_k)$$

If Σ_M is empty then $\tau_\Sigma = \mathbf{1}$. The combined constructor type for Σ, denoted by σ_Σ, is defined by

$$\sigma_\Sigma \quad = \quad \mathcal{T}_C(c_1) + \cdots + \mathcal{T}_C(c_l)$$

under the assumption, that $\Sigma_C = \{c_1, \ldots, c_l\}$. If Σ_C is empty then $\sigma_\Sigma = \mathbf{0}$.

Note that σ_Σ is always a constant type. For τ_Σ we have that

$$\tau_\Sigma \text{ is a} \left\{ \begin{array}{l} \text{polynomial} \\ \text{extended-polynomial} \\ \text{higher-order} \end{array} \right\} \text{type if } \Sigma \text{ contains} \left\{ \begin{array}{l} \text{only polynomial methods} \\ \text{no higher-order methods} \\ \text{a higher-order method.} \end{array} \right.$$

Only practical considerations are responsible for allowing several method and constructor declarations in one class signature. Any class signature Σ is equivalent to a signature Σ' that contains exactly one method declaration of type $\mathsf{Self} \Rightarrow \tau_\Sigma$ and one constructor declaration of type $\sigma_\Sigma \Rightarrow \mathsf{Self}$. So one could equivalently define the term coalgebraic class signature as a pair $\langle \tau, \sigma \rangle$ of an arbitrary type τ and a constant type σ. However, in applications it is nice to have different names for different operations.

From Proposition 4.2.8 we can deduce that $[\![\tau_\Sigma]\!]$ is a polynomial functor if Σ contains only polynomial method declarations and the set of type constructors $\mathcal{C}$ contains only type constants. Similarly for extended polynomial functors and higher-order polynomial functors.

However, it is much more interesting to build class signatures which use type constructors of arity greater than zero in their method declarations, like in the example of queues. Let C be such a type constructor and let Σ be a coalgebraic class signature that makes use of C. In this case the functor $[\![\tau_\Sigma]\!]$ depends in a nontrivial way on the semantics $[\![\mathsf{C}]\!]$ of the type constructor C (which comes along with a model of the ground signature). Because there is no restriction on $[\![\mathsf{C}]\!]$ one cannot say much about the properties of $[\![\tau_\Sigma]\!]$. In a typical application of CCSL all (nonconstant) type constructors stem from abstract data type specifications (to be dealt with in Section 4.6) and from coalgebraic class specifications. In this case $[\![\tau_\Sigma]\!]$ is a data functor in the sense of (Hensel, 1999;

Rößiger, 2000b), provided a technical condition on the variances of type parameters is fulfilled; see Section 4.7.

For a fixed interpretation of the type parameters a model of a coalgebraic class signature Σ is a triple consisting of the state space of the model, a coalgebra for the functor $[\![\tau_\Sigma]\!]$ that interprets the method declarations, and an algebra for the functor $[\![\sigma_\Sigma]\!]$ that is used for the constructor declarations. A complete model is then a collection of such triples indexed by the interpretations for the type parameters.

Definition 4.4.4 (Model of Class Signature) Let Σ be a coalgebraic class signature with n type parameters $\alpha_1, \dots, \alpha_n$. A *model* for Σ consists of an indexed collection of triples $\big(\langle X, c, a\rangle_{U_1,\dots,U_n}\big)_{U_i \in |\mathbf{Set}|}$ where, for each interpretation $U_1, \dots, U_n$ of the type parameters, X is a set (the state space), c is a coalgebra, and a is an algebra as in

$$[\![\sigma_\Sigma]\!]_{U_1,U_1,U_2,U_2,\dots,U_n,U_n} \xrightarrow{\qquad a \qquad} X \xrightarrow{\qquad c \qquad} [\![\tau_\Sigma]\!]_{U_1,U_1,U_2,U_2,\dots,U_n,U_n}(X, X)$$

Remark 4.4.5 The preceding definition does not distinguish between co– and contravariant occurrences of the type variables and of Self. Therefore, a proper model $\mathcal{M}_\Omega$ of the ground signature is not strictly necessary here. It is sufficient if $\mathcal{M}_\Omega$ defines $[\![\mathsf{C}]\!]$ for those argument vectors whose respective co– and contravariant positions are equal.

In case $\mathcal{M}_\Omega$ is proper one can form the following category of signature models for every interpretation $U_1, \dots, U_n$ of the type parameters: Objects are triples $\langle X, c, a\rangle$ and $\langle Y, d, b\rangle$. Morphisms are $[\![\tau_\Sigma]\!]$ coalgebra morphism that commute with the constructors:

$$\begin{tikzcd}
 & X \arrow[r, "c"] \arrow[dd, "f"] & {[\![\tau_\Sigma]\!](X,X)} \arrow[dr, "{[\![\tau_\Sigma]\!](X,f)}"] & \\
{[\![\sigma_\Sigma]\!]} \arrow[ur, "a"] \arrow[dr, "b"'] & & & {[\![\tau_\Sigma]\!](X,Y)} \\
 & Y \arrow[r, "d"] & {[\![\tau_\Sigma]\!](Y,Y)} \arrow[ur, "{[\![\tau_\Sigma]\!](f,Y)}"'] &
\end{tikzcd}$$

Instead of the left triangle one could require that the constructor algebras are behaviourally equivalent, that is, that[6] $\forall u \in [\![\sigma_\Sigma]\!] \,.\, (a\,u)\ {}_{c}\underline{\leftrightarrow}_{d}\ (b\,u)$.

Example 4.4.6 (Model for Queue) The signature Σ_{Queue} from Example 4.4.2 has one type parameter α. A model for this signature consists of a set X_U for every set U, a coalgebra $c_U : X_U \longrightarrow (U \Rightarrow X_U) \times \mathsf{Lift}[U \times X_U]$ and an algebra $a_U : \mathbf{1} \longrightarrow X_U$. To describe such a model let $\mathbb{N}^+$ be the natural numbers including infinity ∞ and take

$$X_U \quad = \quad \{(n, f) \mid n \in \mathbb{N}^+ \wedge f : {<}n \longrightarrow U\}$$

where ${<}n \;=\; \{i \mid i < n\}$ is the initial segment of $\mathbb{N}^+$ below n.[7] So a state in X_U is a pair $\langle n, f\rangle$, consisting of the number n of elements in the queue and a function f that

[6] By anticipating Definition 4.4.7 this condition is equivalent with $\mathrm{Rel}([\![\sigma_\Sigma \Rightarrow \mathsf{Self}]\!])({}_{c}\underline{\leftrightarrow}_{d})(a, b)$.

[7] Note that one cannot use $\mathbb{N}^+ \times (\mathbb{N} \Rightarrow U)$ for X_U, because then it is impossible to define new for $U = \emptyset$.

gives the elements in the queue for arguments less than n. I set

$$\begin{array}{rcll} c(n,f) & = & \begin{cases} \big(\lambda u : U \,.\, (1,\ \lambda i \,.\, u),\quad \mathsf{bot}\big) & \text{if } n = 0 \\ \big(\lambda u : U \,.\, (n, f),\quad \mathsf{up}(f(0),\ (\infty,\ \lambda i \,.\, f(i+1)))\big) & \text{if } n = \infty \\ \big(\lambda u : U \,.\, (n+1,\ \lambda i \,.\ \text{if } i = n \text{ then } u \text{ else } f(i)), & \text{otherwise} \\ \quad \mathsf{up}(f(0),\ (n-1, \lambda n \,.\, f(n+1)))\big) & \end{cases} \\ \mathsf{new} & = & (0, f_\emptyset) \end{array}$$

where $f_\emptyset$ is the empty function $\emptyset \longrightarrow U$. It is easy to see, that the coalgebra c obeys the dependent typing of X_U.[8] Note that at this stage there is nothing that restricts the behaviour of these methods: There exist models of the Queue signature that contain only infinite queues and there are also models that do not resemble FIFO queues at all. ■

There exist class signatures which *do not* have a model. This is because I explicitly allow the empty set as interpretation for type parameters. An example is a class signature $\Sigma_\emptyset$ with one method declaration $m : \mathsf{Self} \Rightarrow \alpha$ and one constructor declaration $c : \mathbf{1} \Rightarrow \mathsf{Self}$. There is no set X with two functions $\mathbf{1} \longrightarrow X \longrightarrow \emptyset$, so there is no model of $\Sigma_\emptyset$ if the type parameter α is interpreted by the empty set. With slight changes the signature $\Sigma_\emptyset$ can be made consistent: either the method declaration is changed to $m : \mathsf{Self} \Rightarrow (\alpha + \mathbf{1})$ or the constructor declaration is changed to $c : \alpha \Rightarrow \mathsf{Self}$.

There are mainly two possibilities to ensure that every signature has a model. First one could restrict the interpretation of the type parameters to nonempty sets. This restriction takes effect if one uses CCSL together with ISABELLE/HOL, because there are no empty types in ISABELLE. A second possibility is to require that all constructors are parametrised by a tuple of all type parameters. This requirement could be combined with an emptiness analysis of the method types.

In the last part of this subsection I explain how a model of a class signature Σ gives rise to an interpretation of the method declarations of Σ and all its subsignatures.

Let $\mathcal{M} = \langle X, c, a \rangle$ be a model of an arbitrary class signature Σ for a fixed interpretation of its type parameters. For every method declaration $m : \tau$ there is a projection

$$\pi_m \ :\ [\![\tau_\Sigma]\!](X,X) = [\![\cdots \times \mathcal{T}_M(m) \times \cdots]\!](X,X) \quad\longrightarrow\quad [\![\mathcal{T}_M(m)]\!]\,(X,X)$$

which extends to a natural transformation $\tau_m : [\![\tau_\Sigma]\!] \Longrightarrow [\![\mathcal{T}_M(m)]\!]$. Similarly for every constructor declaration $e : \tau$ there is an injection

$$\kappa_e \ :\ [\![\mathcal{T}_C(e)]\!] \longrightarrow [\![\cdots + \mathcal{T}_C(e) + \cdots]\!] = [\![\sigma_\Sigma]\!]$$

extending to a natural transformation $\kappa_e : [\![\mathcal{T}_C(e)]\!] \Longrightarrow [\![\sigma_\Sigma]\!]$.

Via these projections and injections the model $\mathcal{M}$ gives rise to an interpretation of the method and constructor declarations. Let $m : \mathsf{Self} \times \sigma_1 \times \cdots \times \sigma_n \Rightarrow \rho$ be a method

[8] The corresponding proofs have all been done in PVS. This was a quite difficult task for PVS, it revealed six bugs, see problem reports number 483–486 on http://pvs.csl.sri.com/cgi-bin/pvs/pvs-bug-list/.

declaration in Σ_M and $e : \sigma \Rightarrow \mathsf{Self}$ be a constructor declaration in Σ_C. Then

$$\begin{array}{rcl} [\![m]\!]^{\mathcal{M}} & = & \lambda x : X, p_1 : [\![\sigma_1]\!], \ldots, p_n : [\![\sigma_n]\!] \,.\, (\pi_m(c\,x))(p_1, \ldots, p_n) \\ [\![e]\!]^{\mathcal{M}} & = & a \circ \kappa_e \end{array}$$

Assume now a subsignature Σ' of Σ and let $\Sigma'_M = \{m_1, \ldots, m_n\}$. The natural transformation

$$\langle \pi_{m_1}, \ldots, \pi_{m_n} \rangle \ : \ [\![\tau_\Sigma]\!] \Longrightarrow [\![\tau_{\Sigma'}]\!]$$

defined by component wise pairing is called the *subsignature projection* and denoted by $\pi_{\Sigma'}$ (where Σ is left implicit). Lemma 3.2.5 (on page 81) shows that $\pi_{\Sigma'}$ gives rise to a functor that maps $[\![\tau_\Sigma]\!]$ coalgebras to $[\![\tau_{\Sigma'}]\!]$ coalgebras. Its object part is given by post composition; for any coalgebra $c : X \longrightarrow [\![\tau_\Sigma]\!]$ there is the following coalgebra for Σ'

$$\pi_{\Sigma'} \circ c \ = \ \langle \pi_{m_1}, \ldots, \pi_{m_n} \rangle \circ c \ : \ X \longrightarrow [\![\tau_{\Sigma'}]\!]$$

This way a model $\mathcal{M}$ for Σ provides an interpretation for the method declarations of all subsignatures of Σ.

4.4.1. Invariants and Bisimulations

Definition 3.3.3 (on page 85) defines bisimulations and invariants for higher-order polynomial functors. This definition can be directly applied to the models of a class signature Σ only under the following condition: The signature Σ contains no type parameters and all type constructors in Σ are type constants. In this restricted case Proposition 4.2.8 applies and further, the interpretation for the type variables is not mixed up with constants.

For the general case it is necessary to extend predicate and relation lifting for type variables and nonconstant type constructors. Let me explain how this works for predicate lifting. Let τ be a type with type variables $\alpha_1, \ldots, \alpha_n$. Fix an interpretation $U_1, \ldots, U_n$ such that U_i is used for the positive and the negative occurrences of α_i in τ. For each type variable the predicate lifting $\mathrm{Pred}([\![\tau]\!])$ gets two additional parameter predicates $P_i^-, P_i^+ \subseteq U_i$. These predicates are used for the negative and positive occurrences of α_i in τ, respectively.[9] For the type constructors that occur in τ one simply uses the predicate lifting that is supplied by the model of the ground signature.

Definition 4.4.7 (Predicate and Relation Lifting) Let $\alpha_1, \ldots, \alpha_n \vdash \tau : \mathsf{Type}$ be a type over an arbitrary proper ground signature Ω. Fix a model $\mathcal{M}$ of Ω, an interpretation $U_1, \ldots, U_n$ for the type variables, and an interpretation X for Self.

[9] One could generalise predicate lifting (and also relation lifting) to take argument predicates $P_i^- \subseteq U_i^-$, $P_i^+ \subseteq U_i^+$, where U_i^- interprets the negative occurrences of α_i and U_i^+ the positive ones. However, predicate lifting is only used within one model where $U_i^- = U_i^+$.

1. The *predicate lifting* of the interpretation of τ (with respect to $\mathcal{M}$), denoted by $\mathrm{Pred}(\llbracket\tau\rrbracket)$, is an operation that takes $2n+2$ predicates as arguments (two predicates for each type variable and two for Self) and yields a predicate on $\llbracket\tau\rrbracket_{U_1,U_1,\dots,U_n,U_n}(X,X)$. Let $\overline{P} = P_1^-, P_1^+, \dots, P_{n+1}^-, P_{n+1}^+$ be a tuple of predicates such that $P_i^-, P_i^+ \subseteq U_i$ for $i \le n$ and $P_{n+1}^-, P_{n+1}^+ \subseteq X$. Let $\overline{P^-} = P_1^+, P_1^-, \dots, P_i^+, P_i^-, \dots, P_{n+1}^+, P_{n+1}^-$ denote the tuple in which the predicates are pairwise swapped (to exchange the predicates for positive and negative occurrences). The predicate lifting $\mathrm{Pred}(\llbracket\tau\rrbracket)$ is an extension of the predicate lifting for higher-order polynomial functors and is defined by induction on the structure of the interpretation functor $\llbracket\tau\rrbracket$.

$$\begin{array}{rcll}
\mathrm{Pred}(\llbracket\alpha_i\rrbracket)(\overline{P}) & = & P_i^+ & \\
\mathrm{Pred}(\llbracket\mathsf{Self}\rrbracket)(\overline{P}) & = & P_{n+1}^+ & \\
\mathrm{Pred}(\llbracket\mathsf{Prop}\rrbracket)(\overline{P}) & = & \top_{\mathsf{bool}} & = \quad \{\top,\bot\} \\
\mathrm{Pred}(\llbracket\mathbf{1}\rrbracket)(\overline{P}) & = & \top_{\mathbf{1}} & = \quad \{*\} \\
\mathrm{Pred}(\llbracket\mathbf{0}\rrbracket)(\overline{P}) & = & \top_{\mathbf{0}} & = \quad \emptyset \\
\mathrm{Pred}(\llbracket\sigma+\tau\rrbracket)(\overline{P}) & = & \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P}) \;+_{\mathrm{P}} \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P}) & \\
 & = & \big\{(\kappa_1\, x) \mid \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P})(x)\big\} \;\cup & \\
 & & \qquad \big\{(\kappa_2\, y) \mid \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P})(y)\big\} & \\
\mathrm{Pred}(\llbracket\sigma\times\tau\rrbracket)(\overline{P}) & = & \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P}) \;\times_{\mathrm{P}} \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P}) & \\
 & = & \big\{(x,y) \mid \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P})(x) \;\;\wedge\;\; \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P})(y)\big\} & \\
\mathrm{Pred}(\llbracket\sigma\Rightarrow\tau\rrbracket)(\overline{P}) & = & \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P^-}) \;\Rightarrow_{\mathrm{P}} \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P}) & \\
 & = & \big\{f \;\mid\; \forall x \in \llbracket\sigma\rrbracket \,.\, \mathrm{Pred}(\llbracket\sigma\rrbracket)(\overline{P^-})(x) & \\
 & & \qquad\qquad\quad \text{implies}\;\; \mathrm{Pred}(\llbracket\tau\rrbracket)(\overline{P})(f\,x)\big\} & \\
\mathrm{Pred}(\llbracket\mathsf{C}[\sigma_1,\dots,\sigma_{\Bbbk}]\rrbracket)(\overline{P}) & = & \llbracket\mathsf{Pred}_{\mathsf{C}}\rrbracket_{A_1,\dots,A_n}\big(\mathrm{Pred}(\llbracket\sigma_1\rrbracket)(\overline{P^-}), \mathrm{Pred}(\llbracket\sigma_1\rrbracket)(\overline{P}), & \\
 & & \qquad\qquad\qquad\qquad \vdots & \\
 & & \qquad\qquad \mathrm{Pred}(\llbracket\sigma_{\Bbbk}\rrbracket)(\overline{P^-}), \mathrm{Pred}(\llbracket\sigma_{\Bbbk}\rrbracket)(\overline{P})\big) &
\end{array}$$

where, in the case for the constructor, $A_i = \llbracket\sigma_i\rrbracket_{\overline{U}}(X,X)$.

2. Fix now a second interpretation $V_1,\dots,V_n,Y$ for the type variables α_i and for Self and let $\llbracket\tau\rrbracket_{\overline{V}}$ and $\llbracket\tau\rrbracket_{\overline{U}}$ denote the interpretation of τ with respect to the V_i and U_i, respectively. Let $\overline{R} = R_1^-, R_1^+, \dots, R_{n+1}^-, R_{n+1}^+$ be a tuple of relations such that $R_i^-, R_i^+ \subseteq U_i \times V_i$ and $R_{n+1}^-, R_{n+1}^+ \subseteq X \times Y$. The *relation lifting* of τ (with respect to the ground signature model $\mathcal{M}$), denoted by $\mathrm{Rel}(\llbracket\tau\rrbracket)$, is an operation that maps the tuple $\overline{R}$ to a relation on $\llbracket\tau\rrbracket_{\overline{U}}(X,X) \times \llbracket\tau\rrbracket_{\overline{V}}(Y,Y)$. It is defined as an

extension of the relation lifting for higher-order polynomial functors by induction on the structure of the interpretation of τ:

$$
\begin{array}{rcll}
\mathrm{Rel}(\llbracket \alpha_i \rrbracket)(\overline{R}) & = & R_i^+ & \\
\mathrm{Rel}(\llbracket \mathsf{Self} \rrbracket)(\overline{R}) & = & R_{n+1}^+ & \\
\mathrm{Rel}(\llbracket \mathsf{Prop} \rrbracket)(\overline{R}) & = & \mathrm{Eq}(\mathsf{bool}) & = \{(a, b) \mid a = b\} \\
\mathrm{Rel}(\llbracket \mathbf{1} \rrbracket)(\overline{R}) & = & \mathrm{Eq}(\mathbf{1}) & = \{(*, *)\} \\
\mathrm{Rel}(\llbracket \mathbf{0} \rrbracket)(\overline{R}) & = & \mathrm{Eq}(\mathbf{0}) & = \emptyset
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{Rel}(\llbracket \sigma + \tau \rrbracket)(\overline{R}) & = & \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R}) \;+_{\mathrm{R}} \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R}) \\
& = & \big\{(\kappa_1\, x_1, \kappa_1\, y_1) \mid \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R})(x_1, y_1)\big\} \;\cup \\
& & \qquad \big\{(\kappa_2\, x_2, \kappa_2\, y_2) \mid \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R})(x_2, y_2)\big\} \\
\mathrm{Rel}(\llbracket \sigma \times \tau \rrbracket)(\overline{R}) & = & \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R}) \;\times_{\mathrm{R}} \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R}) \\
& = & \big\{((x_1, x_2), (y_1, y_2)) \mid \\
& & \quad \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R})(x_1, y_1) \;\wedge\; \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R})(x_2, y_2)\big\} \\
\mathrm{Rel}(\llbracket \sigma \Rightarrow \tau \rrbracket)(\overline{R}) & = & \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R^-}) \;\Rightarrow_{\mathrm{R}} \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R}) \\
& = & \big\{(g, h) \mid \forall x \in \llbracket \sigma \rrbracket_{\overline{U}}(X, X),\; y \in \llbracket \sigma \rrbracket_{\overline{V}}(Y, Y)\;. \\
& & \quad \mathrm{Rel}(\llbracket \sigma \rrbracket)(\overline{R^-})(x, y) \text{ implies } \mathrm{Rel}(\llbracket \tau \rrbracket)(\overline{R})(g\, x, h\, y)\big\} \\
\mathrm{Rel}(\llbracket \mathsf{C}[\sigma_1, \ldots, \sigma_{\Bbbk}] \rrbracket)(\overline{R}) & = & \llbracket \mathsf{Rel}_{\mathsf{C}} \rrbracket_{\overline{A}, \overline{B}}\big(\mathrm{Rel}(\llbracket \sigma_1 \rrbracket)(\overline{R^-}), \mathrm{Rel}(\llbracket \sigma_1 \rrbracket)(\overline{R}), \\
& & \qquad\qquad \vdots \\
& & \qquad \mathrm{Rel}(\llbracket \sigma_{\Bbbk} \rrbracket)(\overline{R^-}), \mathrm{Rel}(\llbracket \sigma_{\Bbbk} \rrbracket)(\overline{R})\big)
\end{array}
$$

where, in the case for the constructor, $\overline{A}$ stands for the list $\llbracket \sigma_i \rrbracket_{\overline{U}}(X, X)$ and $\overline{B}$ for $\llbracket \sigma_i \rrbracket_{\overline{V}}(Y, Y)$.

Remark 4.4.8 The liftings of the preceding definition are sometimes referred to as *full liftings* in contrast to Definition 3.3.1 that neglects type variables. The structure for type variables is not needed for the definition of bisimulation and invariant in this subsection, but it will be needed to give semantics to iterated specifications in Section 4.7. In the preceding definition the cases for constants, Self, product, coproduct, and exponent match exactly Definition 3.3.1.

In this presentation I prefer to consider predicate and relation lifting (and also bisimulations and invariants) completely as semantic notions. One could equivalently define predicate and relation lifting for types as expressions in the logic of CCSL.

The requirement of a proper ground signature in the preceding definition cannot be dropped. If for one type constructor C from Ω its predicate lifting $\mathsf{Pred}_{\mathsf{C}}$ (respectively its relation lifting $\mathsf{Rel}_{\mathsf{C}}$) is not available, then predicate lifting (relation lifting) for types over Ω cannot be defined.

It is possible to adopt the results about predicate and relation lifting of Chapter 3 to the full liftings of the preceding definition. For most of the results one has to assume that the liftings of the involved type constructors have appropriate properties. For the commutation of the liftings with truth and equality (as in Lemma 3.3.2 on page 84) the required properties are built-in into the notion of a proper model of a ground signature.

Lemma 4.4.9 *Let $\mathcal{M}$ be a proper model of a proper ground signature Ω. Then predicate lifting and relation lifting (with respect to $\mathcal{M}$) commute with truth and equality, respectively:*

$$\begin{array}{rcl} \mathrm{Pred}(\llbracket\tau\rrbracket)(\top_{U_1}, \top_{U_1}, \dots, \top_n, \top_n, \top_X, \top_X) & = & \top_{\llbracket\tau\rrbracket} \\ \mathrm{Rel}(\llbracket\tau\rrbracket)(\mathrm{Eq}(U_1), \mathrm{Eq}(U_1), \dots, \mathrm{Eq}(U_n), \mathrm{Eq}(U_n), \mathrm{Eq}(X), \mathrm{Eq}(X)) & = & \mathrm{Eq}(\llbracket\tau\rrbracket) \end{array}$$

where τ is an arbitrary type over Ω with n type variables and $U_1, \dots, U_n, X$ is a (fixed) interpretation of the type variables and of Self.

Proof By induction on the structure of τ, as in Lemma 3.3.2. □

The notions of bisimulation and invariant are defined using predicate and relation lifting. Bisimulations in CCSL are a generalisation of Hermida/Jacobs bisimulations from Definition 3.3.3 (on page 85). However, the invariants in CCSL are the strong invariants from Subsection 3.4.6 (starting on page 103). For class signatures with at least one nonpolynomial method declaration this differs from Hermida/Jacobs invariants of Definition 3.3.3.

Technically a bisimulation should relate two models, so a bisimulation should be a family of relations indexed by the interpretation of the type parameters. This generality is rarely needed: When working with bisimulations one is usually in a context where the interpretation of the type parameters is fixed. Therefore I prefer to define the notions of bisimulation and invariant only for a fixed interpretation of the type parameters.

Definition 4.4.10 (Bisimulation & Invariant) Let Σ be a coalgebraic signature with n type parameters over a proper ground signature Ω. Assume that $\mathcal{M} = \langle X, c, a\rangle$ and $\mathcal{M}' = \langle Y, d, b\rangle$ are models of Σ for a fixed interpretation U_i of the type parameters.

1. A predicate $P \subseteq X$ is an *invariant* for $\mathcal{M}$ if for all $x \in X$

$$P(x) \quad \text{implies} \quad \mathrm{Pred}(\llbracket\tau_\Sigma\rrbracket)(\top_{U_1}, \top_{U_1}, \top_{U_2}, \top_{U_2}, \dots, \top_X, P)(c(x))$$

2. A predicate $P \subseteq X$ *holds initially* in $\mathcal{M}$ if

$$\mathrm{Pred}(\llbracket\sigma_\Sigma \Rightarrow X\rrbracket)(\top_{U_1}, \top_{U_1}, \top_{U_2}, \top_{U_2}, \dots, \top_X, P)(a)$$

3. A relation $R \subseteq X \times Y$ is a *bisimulation* for $\mathcal{M}$ and $\mathcal{M}'$ if for all $x \in X, y \in Y$

$$
\begin{array}{ll}
R(x,y) & \text{implies} \\
& \quad \mathrm{Rel}(\llbracket\tau_\Sigma\rrbracket)(\mathrm{Eq}(U_1), \mathrm{Eq}(U_1), \mathrm{Eq}(U_2), \mathrm{Eq}(U_2), \ldots, R, R)(c(x), d(y))
\end{array}
$$

Note that the notions of bisimulation and invariant are only defined for proper ground signatures. The union of all bisimulations on one model $\mathcal{M}$ for a fixed interpretation of the type parameters is denoted by $\underline{\leftrightarrow}_{\mathcal{M}}$. The fact whether the relation $\underline{\leftrightarrow}_{\mathcal{M}}$ is again a bisimulation depends both on the class signature Σ *and* on the model of the ground signature Ω. If, for instance, the class signature contains only polynomial methods and the ground signature is plain, then bisimilarity $\underline{\leftrightarrow}_{\mathcal{M}}$ is a bisimulation for all proper models of Ω. I discuss the case of a non-plain ground signature in Section 4.7.

For proper models of proper ground signatures it is possible to adopt Proposition 3.3.6 to arbitrary coalgebraic signatures.

Proposition 4.4.11 *Let $\mathcal{M}_\Omega$ be a proper model of a proper ground signature Ω. Then for any model $\mathcal{M}_\Sigma = \langle X, c, a\rangle$ of an arbitrary coalgebraic signature Σ the truth predicate $\top_X$ is an invariant for $\mathcal{M}_\Sigma$ and the equality relation* $\mathrm{Eq}(X)$ *is a bisimulation for $\mathcal{M}_\Sigma$.*

Proof Apply Lemma 4.4.9. □

4.4.2. Class Signatures in CCSL

In CCSL class signatures are part of class specifications. The main difference to Definition 4.4.1 is, that in CCSL type parameters must be declared in advance. An identifier which is neither in the ground signature nor declared as a type parameter yields an error. Figure 4.5 shows the signature of the queues from Example 4.4.2 in CCSL. The grammar for class specifications is as follows:

```
classspec     ::=  BEGIN identifier [ parameterlist ] : [ FINAL ] CLASSSPEC
                   ⦃ importing ⦄ ⦃ classsection ⦄
                   END identifier

classsection  ::=  inheritsection
                |  [ visibility ] attributesection [ ; ]
                |  [ visibility ] methodsection [ ; ]
                |  definitionsection
                |  classconstructorsection [ ; ]
                |  assertionsection
                |  creationsection
                |  theoremsection
                |  requestsection [ ; ]
```

```
Begin Queue[ A : Type ] : ClassSpec
  Method
    put : [Self, A] -> Self;
    top : Self -> Lift[[A, Self]];

  Constructor
    new : Self;
End Queue
```

Figure 4.5.: The queue signature in CCSL syntax

```
visibility    ::=  PUBLIC
                |  PRIVATE
```

Every specification in CCSL starts with the keyword `BEGIN` followed by the name of the specification and the type parameters in brackets. Variance annotations for type parameters are treated like type constraints. If they are present the compiler compares them with the internally computed variances. The compiler reports an error if the variance annotations are too restrictive (i.e., giving a `POS` annotation for a type parameter that has mixed variance). To facilitate aggregation the CCSL compiler generates an axiomatic model for every class specification. The model is either the final one, or an arbitrarily chosen loose one, depending on whether the keyword `FINAL` is present.

A class specification can start with importing clauses (see Subsection 4.9.4 on page 221 below). Importing clauses in class specifications are only needed under special circumstances.

The body of a specification consists of a number of sections. The attribute section, the method section, and the section of class constructors constitute the class signature. The definition section defines definitional extensions. For the inherit section see Subsection 4.8.1. The assertion section, the creation section, and the theorem section are explained in Section 4.5. The assertion section and the creation section contain the axioms of the specification. The theorem section has no influence on the semantics of the specification. It allows the user to exploit the CCSL compiler for translating formulae (which are hopefully provable theorems) into the logic of the target theorem prover. Finally, the request section is there to request the generation of relation liftings for particular types, see Subsection 4.9.3.

The attribute section provides a form of syntactic sugar to ease the modelling of the state of the objects as a record. Formally an attribute declaration is a method declaration with the additional requirement that the type is of the form $\mathsf{Self} \times \sigma \Rightarrow \tau$ where σ and τ are constant types. For every attribute declaration $\mathsf{a} : \mathsf{Self} \times \sigma \Rightarrow \tau$ the compiler adds a method declaration $\mathsf{set\_a} : \mathsf{Self} \times \sigma \times \tau \Rightarrow \mathsf{Self}$ to the signature. The intention is that set_a is the update operation that can be used to change the value of the attribute a. Further, the CCSL compiler generates a number of assertions that describe the behaviour of the update method, see Section 4.5.4.

The modifiers `PUBLIC` and `PRIVATE` classify the methods and attributes into two disjoint sets. If no modifier is present then the methods or attributes are `PUBLIC` by default. The intention is that private methods should not be visible from outside of the class. However, to enforce this one would need existential types (compare (Mitchell and Plotkin, 1988; Abadi and Cardelli, 1996)), which are not present in the theorem provers PVS and ISABELLE. The CCSL compiler uses the public/private classification to derive two signatures from every specification. The first one contains all methods and attributes, the second one is the subsignature containing only the public attributes and methods. All relevant definitions and lemmas are generated twice, first for the full signature, then for public subsignature. This way the user can prove that two models are bisimilar with respect to the public interface, thus ignoring the private methods and attributes. This can be used, for instance, to prove special refinements (Jacobs and Tews, 2001).

The grammar for the sections of attributes, methods, constructors, and definitions is as follows.

```
attributesection         ::= ATTRIBUTE member ⦃ ; member ⦄
methodsection            ::= METHOD member ⦃ ; member ⦄
member                   ::= identifier : type -> type
definitionsection        ::= DEFINING member formula ; ⦃ member formula ; ⦄
classconstructorsection  ::= CONSTRUCTOR classconstructor ⦃ ; classconstructor ⦄
classconstructor         ::= identifier : type
                           | identifier : type -> type
```

Each of these sections contains a list of attributes, methods, constructors, or definitions. The CCSL compiler checks that the declared attributes and methods have method types according to Definition 4.2.7. The constructors must have constant constructor types. The identifiers declared as attributes and methods (together with inherited attributes and methods) form the set of method declarations Σ_M and the class constructor declarations form the set Σ_C.

In the definition section one can define additional methods in terms of other methods (and attributes). The defining formula must be an equation according to the syntax described in Subsection 4.5.4. One can use the full power of CCSL's logic with the following exceptions: Modal operators of the current class and behavioural equality on a type that contains Self are not allowed in definitional extensions. The reason for this restriction is that the CCSL compiler outputs definitions at a position at which these notions are not yet defined for the current class.

For the semantics of the class signatures the CCSL compiler deviates slightly from

```
QueueInterface[Self : Type , A : Type] : Theory
Begin
  Importing Lift[[A , Self]]

  QueueSignature : Type =
    [# put : [[Self , A] -> Self],
       top : [Self -> Lift[[A , Self]]]
    #]
  QueueConstructors : Type = [#    new : Self    #]
End QueueInterface
```

Figure 4.6.: PVS translation of the queue signature

Definition 4.4.4 in two points. First, the functors that give the semantics of types are not present in the output. The CCSL compiler uses type expressions in the logic of PVS or ISABELLE instead. The arguments of the functors become additional theory parameters or type variables (compare the discussion on the representation of functors in PVS in Subsection 2.4.4).

The second point is that a model of a signature consists of two labelled records of functions (instead of a coalgebra/algebra pair)[10]. The first record contains for every method declaration $m_i : \mathsf{Self} \times \sigma \Rightarrow \rho$ a function $[\![\mathsf{Self} \times \sigma \Rightarrow \rho]\!](X, X)$ where X is an additional type parameter that works as a place holder for the state space. Similarly the second labelled record contains a function (or constant) for every constructor declaration. With this different notion of model, the CCSL compiler cannot use the definitions of this section for morphisms, bisimulation, and invariants literally. Instead it uses suitable modifications.

As an illustration I show some parts of the PVS material that the CCSL compiler generates for the queue signature. The material is taken from the file `Queue_basic.pvs`, which was obtained by running the CCSL compiler on the queue signature in Figure 4.5. The complete sources are available in the world wide web, see Appendix A.

The first theory formalises the queue signature, it is shown in Figure 4.6. The name of the theory is QueueInterface, it imports the data type Lift from the (translated) prelude, and declares the method signature and the constructor signature of queue as a labelled records. (For more explanations about the syntax of the PVS specification language see Appendix A.1. The theories that are generated by the CCSL compiler and their contents are described in Subsection 4.9.1 on page 217 below.)

The interface theory is used in the following way. Inhabitants of the type QueueSig-

[10] An earlier version of the compiler implemented Definition 4.4.4 exactly. This often lead to complications with automatic proof strategies.

nature correspond to queue coalgebras. Assume that c has this type (i.e., c is a queue signature model), then one can write put(c)$(\cdots)$ in PVS to get the interpretation of the put method with respect to c.[11]

After the signature the compiler outputs theories for predicate lifting and invariants. These theories are a bit more difficult to understand, because the predicate lifting is generated method wise. This way it can be reused for the (method wise) modal operators of CCSL's logic (they are handled in Subsection 4.5.2 on page 173 below). Therefore I prefer to show the output that is generated for relation lifting and bisimulations.

Predicate and relation lifting is built into the CCSL compiler. It can generate expressions corresponding directly to $\mathrm{Rel}(\llbracket\tau\rrbracket)$. After some experience with the first versions of the CCSL compiler we learned that one usually needs $(c \times d)^* \, \mathrm{Rel}(\llbracket\tau\rrbracket)$ (for two coalgebras c and d). Therefore the current version of the compiler mingles method invocation with relation lifting. The result is not so pleasant from a theoretical point of view, but much easier to use in practice. For the queue signature the PVS theory QueueBisimilarity contains the following.

```
c1 : Var QueueSignature[Self1 , A]
c2 : Var QueueSignature[Self2 , A]

Queue_Rel(c1 , c2) :
  [[[Self1 , Self2] -> bool] -> [[Self1 , Self2] -> bool]] =
  Lambda (R: [[Self1 , Self2] -> bool]) :   Lambda (x1: Self1 , x2: Self2) :
     (Forall (a1: A) : R(put(c1)(x1 , a1) , put(c2)(x2 , a1)))      And
      (Cases top(c1)(x1) OF
         bot   : bot?(top(c2)(x2)),
         up(p0): up?(top(c2)(x2)) And
                 Proj_1(p0) = Proj_1(down(top(c2)(x2))) And
                 R(Proj_2(p0) , Proj_2(down(top(c2)(x2))))
       Endcases )
```

The first two lines declare two queue coalgebras.[12] The coalgebra c1 runs on state space Self1 and c2 on Self2. Then Queue_Rel is defined as a function on Self1 × Self2 ⇒ bool. There are several optimisations built-in into the CCSL compiler that simplify the generated output. For the definition of bisimulation the parameter relations are instantiated with equality. This leads to formulae of the form $\forall a, b : \tau \,.\, a = b \supset \cdots$, which can be simplified into $\forall a : \tau \,.\, (\cdots)[a/b]$. With optimisations turned off the fourth line of the definition of Queue_Rel would look as follows

```
Forall(a1 : A, a2 : A) : a1 = a2   Implies   R(put(c1)(x1, a1), put(c2)(x2, a2))
```

[11] In PVS record selection can be written like function application, so put(c) denotes the put field of the record c.

[12] Variable declarations are syntactic sugar in PVS. The two declarations save the lambda abstractions in the definition of Queue_Rel for the arguments c1 and c2. See also the explanation on page 265.

Another optimisation that is performed by the compiler is the inlining of liftings for nonrecursive data types and class types. In the queue example the type constructor Lift is defined as a nonrecursive abstract data type. Therefore the compiler outputs a case expression instead of applying the relation lifting for Lift.

After the relation lifting the compiler outputs a recogniser on queue bisimulations:

```
bisimulation?(c1 , c2) : [[[Self1 , Self2] -> bool] -> bool] =
  Lambda (R: [[Self1 , Self2] -> bool]) :  Forall (x1: Self1 , x2: Self2) :
      R(x1 , x2)    Implies  Queue_Rel(c1 , c2)(R)(x1 , x2)
```

The predicate bisimulation? holds for a relation R if and only if R is a queue bisimulation. With it, bisimilarity is defined as follows.

```
bisim?(c1 , c2) : [[Self1 , Self2] -> bool] =
  Lambda (x1: Self1 , x2: Self2) :    Exists (R: [[Self1 , Self2] -> bool]) :
      bisimulation?(c1 , c2)(R)    And    R(x1 , x2)
```

The generation of definitions for bisimulation and invariant for coalgebraic signatures is only one part of a translation into higher order logic. An equally well important task is the generation of lemmas that capture standard results. For instance, for signatures corresponding to polynomial functors the compiler generates lemmas stating that bisimilarity is an equivalence relation. To achieve this, bisimulation and bisimilarity is first defined for one coalgebra (the following material is from the theories QueueBisimilarityEquivalence and QueueBisimilarityEqRewrite):

```
c : Var QueueSignature[Self , A]
bisimulation?(c) : [[[Self , Self] -> bool] -> bool] = bisimulation?(c , c);
bisim?(c) : [[Self , Self] -> bool] = bisim?(c , c) ;
```

Then, the statement that equality is a queue bisimulation reads as follows:

```
eq_bisim : Lemma bisimulation?(c)(Lambda(x1: Self , x2: Self) : x1 = x2)
```

For reflexivity and symmetry of bisimilarity the compiler generates

```
bisim_refl : Lemma Forall (x: Self) : bisim?(c)(x , x)
bisim_sym : Lemma Forall (x1: Self , x2: Self) :
    bisim?(c)(x1 , x2)   Implies  bisim?(c)(x2 , x1)
```

In reasoning with bisimulations one often needs lemmas that express that a method delivers the same (or bisimilar) results when invoked for bisimilar states. The CCSL compiler generates one such lemma for each method, here I show only the one for the method put.

```
bisim_put : Lemma Forall (x1: Self , x2: Self , a1: A) :
  bisim?(c)(x1 , x2)   Implies  bisim?(c)(put(c)(x1 , a1) , put(c)(x2 , a1))
```

Lemmas like bisim_put seem to be trivial, because they follow immediately from the definition of bisimulation. However, these little lemmas are extremely useful in applications, their generation is one of the great benefits of the CCSL compiler.

Ideally one would like that for all generated lemmas the CCSL compiler outputs proofs in the format of the target theorem prover. However, it is very difficult to generate proofs that work properly for all possible signatures. The current compiler version generates only a few proofs.

4.5. Assertions and Creation Conditions

The previous section discussed the structural aspect of coalgebraic specification. In this section I turn to the logical aspects. The signature of FIFO queues in Example 4.4.2 contains nothing to actually restrict the class of models to those that can be considered as FIFO queues. And, indeed, also (*last–in–first–out*) stacks give rise to models of Σ_{Queue}. In this section I define a logic that allows one to express properties of methods and constructors from a coalgebraic class signature. A signature together with a set of logical formulae is called a specification. The models of the specification are those models of the signature that fulfil the formulae in a suitable sense.

In the following I present the logic of CCSL. This is an entirely standard higher-order logic over a polymorphic signature with two extensions. The extensions are behavioural equality and (infinitary) method-wise modal operators. The first subsection presents the higher-order logic with behavioural equality over a coalgebraic class signature. The second subsection defines infinitary modal operators for coalgebras. Subsection 4.5.3 is on coalgebraic class specifications and Subsection 4.5.4 explains the syntax of CCSL.

4.5.1. Higher-order Logic

The striking property of higher-order logic is that formulae are terms of the special type Prop. Thereby it is possible to quantify over subsets of individuals and also over predicates. Terms may contain *(term) variables*, which are declared to have a certain type in the *term variable context*. All types can contain type variables drawn from a type variable context. Formally a term variable context (over a type variable context Ξ) is a finite list of distinct variable declarations $x : \tau$ such that $\Xi \vdash \tau : \mathsf{Type}$ is derivable.[13] Terms are formed from variables, constructions like tuples or case analysis, and logical connectives. A term is given by a *term judgement*

$$\Xi \mid \Gamma \vdash t : \tau$$

Here Ξ is a type variable context, Γ is a term variable context and t is a well-typed term of type τ according to the rules below. All free (term) variables of t must be declared in Γ and all type variables that occur in t, τ, and Γ must be declared in Ξ.

[13] The condition that a context contains no variable twice can be enforced at the expense of a more complicated derivation system, see for instance Section 2.1 in (Jacobs, 1999a).

The following definition describes the terms and formulae over a coalgebraic class signature. Modal operators are added in Definition 4.5.7 (on page 174) below.

Definition 4.5.1 (Terms and Formulae) Let Σ be a coalgebraic class signature over a proper ground signature Ω. The set of terms over Σ, denoted with $\mathit{Terms}(\Sigma)$, is the least set containing:

- $x : \tau$ for a variable x of type τ
- $* : \mathbf{1}$ the only inhabitant of $\mathbf{1}$
- $\bot : \mathsf{Prop}$, $\top : \mathsf{Prop}$ the boolean constants false and true
- $f : \sigma$ for constants $f \in \Omega_\sigma$
- $m : \mathsf{Self} \times \sigma \Rightarrow \rho$ for all method declarations $m \in \Sigma_M$
- $c : \sigma \Rightarrow \mathsf{Self}$ for all constructor declarations $c \in \Sigma_C$
- $(t_1, t_2) : \sigma \times \tau$, the tuple for terms $t_1 : \sigma$ and $t_2 : \tau$
- $\pi_1\, t : \sigma$ and $\pi_2\, t : \tau$, the projections for a term $t : \sigma \times \tau$
- $\kappa_1\, s : \sigma + \tau$ and $\kappa_2\, t : \sigma + \tau$, the injections for terms $s : \sigma$ and $t : \tau$
- $\mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : r,\ \kappa_2\, y : s \ \ : \ \tau$, the case analyses for terms $t : \sigma_1 + \sigma_2$, $r : \tau$, and $s : \tau$. The term r contains the variable x free and the term s contains y free. The term t gets bound to either x or y, depending on the result of the evaluation. In the complete case expression the variables x and y are bound.
- $\mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t \ : \ \tau$, the conditional for a term $r : \mathsf{Prop}$ and two terms s and t of the same type τ
- $\lambda x : \sigma \,.\, t \ : \ \sigma \Rightarrow \tau$, lambda abstraction for a variable $x : \sigma$ and a term $t : \tau$.
- $t_1\, t_2 : \tau$, application for two terms $t_1 : \sigma \Rightarrow \tau$ and $t_2 : \sigma$
- $t_1 = t_2 : \mathsf{Prop}$ equality for two terms of the same type τ
- $t_1 \sim t_2 : \mathsf{Prop}$, behavioural equality for two terms of the same type τ
- $\neg t : \mathsf{Prop}$, the negation for a term $t : \mathsf{Prop}$,
- $t_1 \wedge t_2 \ : \ \mathsf{Prop}$ and $t_1 \vee t_2 \ : \ \mathsf{Prop}$, the conjunction and the disjunction for terms $t_1 : \mathsf{Prop}$ and $t_2 : \mathsf{Prop}$
- $\forall x : \tau \,.\, t : \mathsf{Prop}$, universal quantification for a variable $x : \tau$ and a term $t : \mathsf{Prop}$

A derivation system for term judgements for well-typed terms is in the Figures 4.7 and 4.8. As abbreviations I define

- $t_1 \supset t_2 \overset{\text{def}}{=} \neg t_1 \vee t_2$, implication
- $t_1 \supset\!\!\subset t_2 \overset{\text{def}}{=} (t_1 \supset t_2) \wedge (t_2 \supset t_1)$, logical equivalence
- $\mathsf{let}\ x : \tau = t_1\ \mathsf{in}\ t_2 \overset{\text{def}}{=} (\lambda x : \tau \,.\, t_2)\, t_1$, let bindings
- $\exists x : \tau \,.\, t \overset{\text{def}}{=} \neg \forall x : \tau \,.\, \neg t$, existential quantification

Terms of type Prop are called formulae and denoted with Greek letters like φ, ψ. The formulae over Σ are denoted with $\mathit{Form}(\Sigma)$.

The only non-standard term in the preceding definition is behavioural equality. Its semantics is given by the relation lifting of bisimilarity. For instance for two terms t_1 and t_2 of type $\alpha \times \mathsf{Self}$ the equation $t_1 \sim t_2$ holds if and only if $\pi_1\, t_1 = \pi_1\, t_2$ and $(\pi_2\, t_1) \; \underline{\leftrightarrow} \; (\pi_2\, t_2)$. For class signatures over non-proper ground signatures one has to restrict the terms to those which do not contain behavioural equality $\sim$.

Example 4.5.2 In Example 4.4.2 I described the Queue signature. Here I show two formulae that separate *FIFO* queues from other models of the Queue–signature. The first property is about empty queues. A queue q is considered empty if the top methods fails on it (i.e., if $\mathsf{top}(q) = \mathsf{bot}$).

$$F_{\mathsf{empty}}(q) \quad \overset{\text{def}}{=} \quad \Big[\quad \mathsf{top}(q) = \mathsf{bot} \quad \supset \quad \forall a : \alpha \,.\, \mathsf{top}(\mathsf{put}(q, a)) \sim \mathsf{up}(a, q) \quad \Big]$$

So if the queue is empty then $\mathsf{top}(\mathsf{put}(q, a))$ should always be successful (i.e., it never equals bot) and return a pair (b, q') where $a = b$ and q' is an empty queue again. To be precise F_{empty} is a term

$$\alpha : \mathsf{Type} \ \mid \ q : \mathsf{Self} \ \vdash \ F_{\mathsf{empty}}(q) \ : \ \mathsf{Prop}$$

The second property (over the same contexts) is about nonempty queues.

$$F_{\mathsf{filled}}(q) \quad \overset{\text{def}}{=} \quad \left[\begin{array}{l} \forall a_1 : \alpha, q' : \mathsf{Self} \,.\, \mathsf{top}(q) \sim \mathsf{up}(a_1, q') \ \supset \\ \qquad\qquad\qquad \forall a_2 : \alpha \,.\, \mathsf{top}(\mathsf{put}(q, a_2)) \ \sim \ \mathsf{up}(a_1, \mathsf{put}(q', a_2)) \end{array} \right]$$

This says that, if q is nonempty, then the two operations of adding an element (at the end) and of removing the first element are interchangeable. ■

The semantics of the logic is completely standard. Let $\Xi \mid \Gamma \vdash t : \tau$ be a term. Fix an interpretation $U_1, \dots, U_n$ for the type variables in Ξ and a set X as interpretation of Self. You can think of the term t as a function that maps any values that you assign to the

ground terms

$$\frac{\Xi \vdash \sigma : \mathsf{Type}}{\Xi \mid \Gamma \vdash x : \sigma}\ x : \sigma \in \Gamma \qquad \frac{\Xi \vdash \sigma : \mathsf{Type}}{\Xi \mid \Gamma \vdash f : \sigma}\ f \in \Omega_\sigma$$

$$\frac{\Xi \vdash \tau : \mathsf{Type}}{\Xi \mid \Gamma \vdash m : \tau}\ m : \tau \in \Sigma_M \qquad \frac{\Xi \vdash \tau : \mathsf{Type}}{\Xi \mid \Gamma \vdash c : \tau}\ c : \tau \in \Sigma_C$$

$$\frac{}{\Xi \mid \Gamma \vdash * : \mathbf{1}} \qquad \frac{}{\Xi \mid \Gamma \vdash \bot : \mathsf{Prop}} \qquad \frac{}{\Xi \mid \Gamma \vdash \top : \mathsf{Prop}}$$

tuples

$$\frac{\Xi \mid \Gamma \vdash s : \sigma \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash (s, t) : \sigma \times \tau} \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \times \tau}{\Xi \mid \Gamma \vdash \pi_1\, t : \sigma} \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \times \tau}{\Xi \mid \Gamma \vdash \pi_2\, t : \tau}$$

variants

$$\frac{\Xi \mid \Gamma \vdash s : \sigma \quad \Xi \vdash \tau : \mathsf{Type}}{\Xi \mid \Gamma \vdash \kappa_1\, s : \sigma + \tau} \qquad \frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash \kappa_2\, t : \sigma + \tau}$$

$$\frac{\Xi \mid \Gamma \vdash t : \sigma_1 + \sigma_2 \quad \Xi \mid \Gamma, x : \sigma_1 \vdash r : \tau \quad \Xi \mid \Gamma, y : \sigma_2 \vdash s : \tau}{\Xi \mid \Gamma \vdash \mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : r,\ \kappa_2\, y : s \quad : \tau}\ x \notin \Gamma, y \notin \Gamma$$

conditional

$$\frac{\Xi \mid \Gamma \vdash r : \mathsf{Prop} \quad \Xi \mid \Gamma \vdash s : \tau \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash \mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t \ : \ \tau}$$

abstraction & application

$$\frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi \mid \Gamma, x : \sigma \vdash t : \tau}{\Xi \mid \Gamma \vdash \lambda x : \sigma \,.\, t : \sigma \Rightarrow \tau}\ x \notin \Gamma \qquad \frac{\Xi \mid \Gamma \vdash t : \sigma \Rightarrow \tau \quad \Xi \mid \Gamma \vdash s : \sigma}{\Xi \mid \Gamma \vdash t\, s : \tau}$$

Figure 4.7.: Derivation system for the terms over a coalgebraic class signature Σ and a ground signature Ω, Part I

equality

$$\frac{\Xi \mid \Gamma \vdash s : \tau \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash s = t : \mathsf{Prop}} \qquad \frac{\Xi \mid \Gamma \vdash s : \tau \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma \vdash s \sim t : \mathsf{Prop}}$$

conjunction & disjunction

$$\frac{\Xi \mid \Gamma \vdash s : \mathsf{Prop} \quad \Xi \mid \Gamma \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash s \wedge t : \mathsf{Prop}} \qquad \frac{\Xi \mid \Gamma \vdash s : \mathsf{Prop} \quad \Xi \mid \Gamma \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash s \vee t : \mathsf{Prop}}$$

negation **universal quantification**

$$\frac{\Xi \mid \Gamma \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash \neg t : \mathsf{Prop}} \qquad \frac{\Xi \vdash \tau : \mathsf{Type} \quad \Xi \mid \Gamma, x : \tau \vdash t : \mathsf{Prop}}{\Xi \mid \Gamma \vdash \forall x : \tau \,.\, t : \mathsf{Prop}} \; x \notin \Gamma$$

The following rules can be derived.

weakening

$$\frac{\Xi \mid \Gamma \vdash t : \tau}{\Xi, \alpha : \mathsf{Type} \mid \Gamma \vdash t : \tau} \; \alpha \notin \Xi \qquad \frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma, x : \sigma \vdash t : \tau} \; x \notin \Gamma$$

type substitution

$$\frac{\Xi \vdash \sigma : \mathsf{Type} \quad \Xi, \alpha : \mathsf{Type} \mid \Gamma \vdash t : \tau}{\Xi \mid \Gamma[\sigma/\alpha] \vdash t[\sigma/\alpha] : \tau[\sigma/\alpha]} \; \alpha \notin \Xi$$

term substitution

$$\frac{\Xi \mid \Gamma \vdash s : \sigma \quad \Xi \mid \Gamma, x : \sigma \vdash t : \tau}{\Xi \mid \Gamma \vdash t[s/x] : \tau} \; x \notin \Gamma$$

Figure 4.8.: Derivation system for the terms over a coalgebraic class signature Σ and a ground signature Ω, Part II

variables in Γ to a value in τ. Consequently the semantics of t for a fixed interpretation of the type variables and of Self is a function

$$\llbracket \sigma_1 \rrbracket \times \cdots \times \llbracket \sigma_k \rrbracket \longrightarrow \llbracket \tau \rrbracket$$

where I assume that $\Gamma = x_1 : \sigma_1, \ldots, x_k : \sigma_k$. The complete semantics for t is an indexed collection of such functions:

$$\left(\begin{array}{l} \llbracket \sigma_1 \rrbracket_{U_1,U_1,\ldots,U_n,U_n}(X,X) \times \cdots \times \llbracket \sigma_k \rrbracket_{U_1,U_1,\ldots,U_n,U_n}(X,X) \\ \qquad\qquad\qquad\qquad \longrightarrow \llbracket \tau \rrbracket_{U_1,U_1,\ldots,U_n,U_n}(X,X) \end{array} \right)_{X, U_1, \ldots, U_n \in |\mathbf{Set}|}$$

If $t : \tau$ is a formula then $\llbracket \tau \rrbracket$ is the set of booleans and the interpretation function returns true for exactly those elements of $\llbracket \sigma_i \rrbracket$ that fulfil t. So one can equivalently consider the semantics of a formula $x : \sigma \vdash \varphi : \mathsf{Prop}$ as a (collection of) predicate(s) $\llbracket \varphi \rrbracket \subseteq \llbracket \sigma \rrbracket$.

Definition 4.5.3 (Semantics) Let Σ be a coalgebraic class signature over a proper ground signature Ω and assume a model $\mathcal{M}_\Omega$ of Ω and a model $\mathcal{M}_\Sigma = \langle X, c, a \rangle$ of Σ. The interpretation of a term $\alpha_1, \ldots, \alpha_n \mid x_1 : \sigma_1, \ldots, x_k : \sigma_k \vdash t : \tau$ with respect to $\mathcal{M}_\Omega$ and $\mathcal{M}_\Sigma$ is denoted by $\llbracket t \rrbracket^{\mathcal{M}_\Omega, \mathcal{M}_\Sigma}$, where I omit the superscripts if they are clear from the context. The interpretation $\llbracket t \rrbracket$ is defined by induction on the structure of terms. Fix an interpretation $U_1, \ldots, U_n$ of the type variables α_i and and an interpretation X of Self. Let $\overline{x} : \overline{\sigma}$ denote the tuple of arguments $x_1 : \llbracket \sigma_1 \rrbracket, \ldots, x_k : \llbracket \sigma_k \rrbracket$.

$$\begin{array}{rcl}
\llbracket x_i \rrbracket & = & \pi_i \\
\llbracket f \rrbracket & = & \llbracket f \rrbracket \qquad\qquad\qquad\qquad \text{for } f \in \Omega_\sigma \\
\llbracket m : \mathsf{Self} \times \sigma' \Rightarrow \tau' \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, \big(\lambda x : X, p : \llbracket \sigma' \rrbracket \,.\, \pi_m(c\,x)(p) \big) \\
\llbracket c \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, a \circ \kappa_c \\
\llbracket * \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, * \\
\llbracket \bot \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, \bot \\
\llbracket \top \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, \top \\
\llbracket (t_1, t_2) \rrbracket & = & \langle \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket \rangle \\
\llbracket \pi_1\, t \rrbracket & = & \pi_1 \circ \llbracket t \rrbracket \\
\llbracket \pi_2\, t \rrbracket & = & \pi_2 \circ \llbracket t \rrbracket \\
\llbracket \kappa_1\, s \rrbracket & = & \kappa_1 \circ \llbracket t \rrbracket \\
\llbracket \kappa_2\, t \rrbracket & = & \kappa_2 \circ \llbracket t \rrbracket \\
\llbracket \mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : r,\ \kappa_2\, y : s \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, \left\{ \begin{array}{lll} \llbracket r \rrbracket(\overline{x}, z_1) & \text{if} & \llbracket t \rrbracket\, \overline{x} = \kappa_1 z_1 \\ \llbracket s \rrbracket(\overline{x}, z_2) & \text{if} & \llbracket t \rrbracket\, \overline{x} = \kappa_2 z_2 \end{array} \right. \\
\llbracket \mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, \left\{ \begin{array}{lll} \llbracket s \rrbracket \overline{x} & \text{if} & \llbracket r \rrbracket \overline{x} = \top \\ \llbracket t \rrbracket \overline{x} & \text{if} & \llbracket r \rrbracket \overline{x} = \bot \end{array} \right. \\
\llbracket \lambda x : \rho \,.\, t \rrbracket & = & \lambda \overline{x} : \overline{\sigma} \,.\, \big(\lambda y : \llbracket \rho \rrbracket \,.\, \llbracket t \rrbracket(\overline{x}, y) \big)
\end{array}$$

$$
\begin{array}{rcl}
[\![t_1\, t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!](\overline{x})\ ([\![t_2]\!]\,\overline{x}) \\
[\![t_1 = t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!]\overline{x}\ =\ [\![t_2]\!]\overline{x} \\
[\![t_1 \sim t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, \mathrm{Rel}([\![\rho]\!])(\underline{\leftrightarrow}_{\mathcal{M}_\Sigma})([\![t_1]\!]\,\overline{x},\ [\![t_2]\!]\,\overline{x}) \\
 & & \hfill \text{(for } t_1 \text{ and } t_2 \text{ of type } \rho\text{)} \\
[\![\neg t]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, \neg[\![t]\!]\overline{x} \\
[\![t_1 \wedge t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!]\overline{x}\ \wedge\ [\![t_2]\!]\overline{x} \\
[\![t_1 \vee t_2]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, [\![t_1]\!]\overline{x}\ \vee\ [\![t_2]\!]\overline{x} \\
[\![\forall x : \tau \,.\, t]\!] & = & \lambda \overline{x} : \overline{\sigma} \,.\, \begin{cases} \top & \text{if} \quad [\![t]\!](\overline{x}, y) = \top \text{ for all } y \in [\![\tau]\!] \\ \bot & \text{otherwise} \end{cases}
\end{array}
$$

In the following I elaborate on the expressiveness of the logic of CCSL.

Proposition 4.5.4 *The logic of* CCSL *is complete with respect to bisimilarity. More precisely, for every closed term $t \in Term(\Sigma)$ of type* Self *over a signature Σ, there exists a formula $x : \mathsf{Self} \vdash F(x) : \mathsf{Prop}$ with the following property. For an arbitrary model $\mathcal{M} = \langle X, c, a\rangle$ of Σ and a state $x \in X$ one has $[\![F]\!](x) = \top$ if and only if there is a bisimulation on $\mathcal{M}$ relating x and $[\![t]\!]^{\mathcal{M}}$.*

Proof (Sketch) The definition of relation lifting and bisimulation can be directly expressed in the logic of CCSL. Thus there is a formula $R : \mathsf{Self} \times \mathsf{Self} \Rightarrow \mathsf{Prop} \vdash \mathsf{bisim}(R) : \mathsf{Prop}$ which is true, precisely if R is interpreted with a bisimulation. One can take as F

$$\lambda x : \mathsf{Self}\ .\ \exists R : \mathsf{Self} \times \mathsf{Self} \Rightarrow \mathsf{Prop}\ .\ \mathsf{bisim}(R)\ \wedge\ R(t, x) \qquad \square$$

The logic of CCSL has equality on all types, including Self. Therefore it is easy to construct a formula that evaluates to different values for bisimilar states. For coalgebraic specification it is often desirable to restrict the expressiveness of the logic, such that it cannot distinguish bisimilar states, or, in other words, is sound with respect to bisimilarity. This restricted expressiveness is for instance necessary for some results in Section 4.7 and for the result about behavioural refinement in (Jacobs and Tews, 2001). In the following I characterise a fragment of the logic of CCSL that is sound with respect to bisimilarity. The higher-order aspects make it necessary to consider terms in general.

Definition 4.5.5 (Behavioural invariance) Let $\mathcal{M}$ be a model of a proper ground signature Ω and let Σ be a coalgebraic class signature over Ω. Let $\Xi \mid \Gamma \vdash t : \tau$ be a term over Σ with $\Gamma = a_1 : \sigma_1, \ldots, a_k : \sigma_k$. The term t is *invariant with respect to behavioural equality for $\mathcal{M}$* (or more succinctly *behaviourally invariant for $\mathcal{M}$*) if for all models $\mathcal{A} = \big(\langle X, c, a\rangle_{\overline{U}}\big)$ and $\mathcal{B} = \big(\langle Y, d, b\rangle_{\overline{U}}\big)$ and all interpretations $\overline{U} = U_1, \ldots, U_n$ of the type parameters of Σ the following condition is fulfilled. Let $x_i \in [\![\sigma_i]\!]_{\overline{U}}(X, X)$ and $y_i \in [\![\sigma_i]\!]_{\overline{U}}(Y, Y)$ be two interpretations of the variables a_i and let $R \subseteq X \times Y$ be a bisimulation for c and d. If

$$\mathrm{Rel}\big([\![\sigma_i]\!]\big)\ \big(\mathrm{Eq}(U_1), \ldots, \mathrm{Eq}(U_n), R, R\big)\ (x_i, y_i)$$

holds for all i, then it also holds that

$$\mathrm{Rel}\big(\llbracket\tau\rrbracket\big)\ \big(\mathrm{Eq}(U_1),\ldots,\mathrm{Eq}(U_n),R,R\big)\ \big(\ \llbracket t\rrbracket^{\mathcal{M},A}(x_1,\ldots,x_k),\ \llbracket t\rrbracket^{\mathcal{M},B}(y_1,\ldots,y_k)\ \big)$$

This definition is carefully formulated to apply to arbitrary signatures, for which a greatest bisimulation might not exist. Of course, if bisimilarity as greatest bisimulation does exists, then behavioural invariance with respect to bisimilarity implies behavioural invariance with respect to any other bisimulation. It is easy to give some sufficient syntactical criteria for behavioural invariance of terms.

Proposition 4.5.6 *Let Σ be a coalgebraic class signature. The following basic terms are behaviourally invariant:*

- *variables $x : \tau$*
- *the constants $\bot, \top : \mathsf{Prop}$, and $* : \mathbf{1}$*
- *methods from Σ*

The following constructions preserve behavioural invariance:

- *pairing (t_1, t_2): that is if t_1 and t_2 are behavioural invariant then so is (t_1, t_2),*
- *projections $\pi_{1/2}\, t : \sigma$, injections $\kappa_{1/2}\, t$, and case analyses* $\mathsf{cases}\ t\ \mathsf{of}\ \kappa_1\, x : t_1,\ \kappa_2\, y : t_2\ : \tau$
- *the conditional* $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$,
- *lambda abstraction $\lambda x : \sigma\,.\,t$ and application $t_1\, t_2$*
- *negation $\neg t$, conjunction $t_1 \wedge t_2$, and disjunction $t_1 \vee t_2$*

For proper models of the ground signature:

- *universal quantification over constants $\forall x : \tau\,.\,t$, where τ is a constant type (i.e., $\mathcal{V}_{\mathsf{Self}}(\tau) = ?$)*

If additionally bisimulations are closed under composition and if the greatest bisimulation $\underline{\leftrightarrow}$ does exist:

- *behavioural equality $t_1 \sim t_2$*

Proof I abbreviate the longish $\mathrm{Rel}(\llbracket\tau\rrbracket)(\mathrm{Eq}(U_1),\ldots,\mathrm{Eq}(U_n),R,R)$ as $\mathrm{Rel}(\tau)(\cdots)$ and use $\overline{x} = x_1,\ldots,x_k$ and $\overline{y} = y_1,\ldots,y_k$.

- $\mathrm{Rel}(\tau)(\cdots)(\llbracket x\rrbracket^A, \llbracket x\rrbracket^B)$ follows from the definition.

- The basic terms $*$, $\bot$, and $\top$ are behavioural invariant because Definition 4.4.7 (2), uses equality for Prop and $\mathbf{1}$.

- Let $t = m$ be a method, then $\mathrm{Rel}(\tau)(\cdots)([\![m]\!]^{\mathcal{A}}, [\![m]\!]^{\mathcal{B}})$ follows from the definition of bisimulation.

- If t is a pair, a projection, an injection, or a case analyses, then the conclusion follows directly from the definition of relation lifting.

- For the conditional $t = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$ assume that t_1, t_2, and t_3 are behavioural invariant. Then $[\![t_1]\!]^{\mathcal{A}}\,\overline{x} = [\![t_1]\!]^{\mathcal{B}}\,\overline{y}$ and the conclusion follows by the induction hypothesis on t_2 and t_3.

- For $t = \lambda x : \sigma \,.\, t_1$ we have to show that for all $x \in [\![\sigma]\!]^{\mathcal{A}}$ and $y \in [\![\sigma]\!]^{\mathcal{B}}$ with $\mathrm{Rel}(\sigma)(\cdots)(x, y)$ also $\mathrm{Rel}(\tau)(\cdots)([\![t_1]\!]^{\mathcal{A}}(\overline{x}, x), [\![t_1]\!]^{\mathcal{B}}(\overline{y}, y))$. This fact follows directly from the behavioural invariance of t_1.

  The case $t = t_1\, t_2$ follows directly from the behavioural invariance of t_1 and t_2.

- If t is one of the propositional connectives, then the conclusion follows again from the fact that the relation lifting for type Prop is given by equality.

- If $t = \forall a : \sigma \,.\, t'$ then we have $[\![\sigma]\!]^{\mathcal{A}} = [\![\sigma]\!]^{\mathcal{B}}$ because σ is a constant type. Additionally Lemma 4.4.9 yields $\mathrm{Rel}(\sigma)(\cdots) = \mathrm{Eq}([\![\sigma]\!])$ for a proper model of the ground signature. So the assumption about the behavioural invariance of t_1 implies for every $a \in [\![\sigma]\!]$ that $[\![t]\!]^{\mathcal{A}}(\overline{x}, a) = [\![t]\!]^{\mathcal{B}}(\overline{y}, a)$.

- For $t = \ t_1 \sim t_2$ assume that t_1 and t_2 are behavioural invariant. With composition of bisimulations $\mathrm{Rel}(\tau)(\cdots)([\![t_1]\!]^{\mathcal{A}}, [\![t_2]\!]^{\mathcal{A}})$ holds precisely if $\mathrm{Rel}(\tau)(\cdots)([\![t_1]\!]^{\mathcal{B}}, [\![t_2]\!]^{\mathcal{B}})$ holds. Hence $[\![t]\!]^{\mathcal{A}} = [\![t]\!]^{\mathcal{B}}$. □

The notion of behavioural invariance and the preceding proposition are interesting for class signatures for which coalgebra morphisms are functional bisimulations. In this case behavioural invariance implies stability under coalgebra morphisms. I exploit this fact in Proposition 4.5.18 and in Subsection 4.7.2.

Note that the preceding proposition does neither include constants from the ground signature nor constructors from class specifications. Monomorphic constants (i.e., those over the empty type variable context) are behaviourally invariant if the relation lifting for their type coincides with equality (which holds for proper models of proper ground signatures). Polymorphic constants might or might not be behaviourally invariant, depending on the model of the ground signature. With the `-pedantic` switch (see Subsection 4.9.9 on page 223 below) the CCSL compiler recognises polymorphic constants as behavioural invariant if they are instantiated with a constant type. This relaxed policy rests on an argument similar to that in the preceding proof for the item of universal quantification and on the fact that the `-pedantic` switch implies a proper ground signature with a proper model.

4.5.2. Infinitary Modal Operators

This subsection describes joint work with Bart Jacobs and Jan Rothe. Following the observations that, firstly, modal logic (Goldblatt, 1992) is the logic for describing dynamic systems and that, secondly, coalgebras are the mathematical structures that capture dynamic systems one has to expect a close relationship between modal logic and coalgebras. Currently modal logic is used in the field of coalgebras mainly in two different ways. On the one hand modal logic is used as a tool to investigate the theory of coalgebras. On the other hand modal logic enriches coalgebraic specification.

In the former line of work (Moss, 1999) describes characterising (modal) formulae for the state space of a coalgebra. Rößiger uses modal logic to construct final coalgebras for data functors (Rößiger, 2000a) (but see also (Rößiger, 2000b)). Modal logic also plays an important role in the search for the coalgebraic analogy of Birkhoffs theorem (Kurz, 2000; Hughes, 2001; Goldblatt, 2001a). The modal logics of Rößiger, Moss, and Goldblatt are all sound and complete with respect to bisimilarity (i.e., bisimilar states fulfil the same set of formulae and for any two non-bisimilar states there is a formula that distinguishes both). The logics of Kurz and Hughes have this property when restricted to one colour. However, all these logics have often been designed towards a certain theorem. Without deprecating all this work, one notices that these logics are not very practical for expressing interesting properties of coalgebras. For instance in the queue example of this chapter, I consider the following property F_{finite} as interesting: A queue fulfils F_{finite} if the successive application of the method top eventually yields an empty queue (i.e., top returns eventually bot). To express F_{finite} in the framework of (Moss, 1999) or (Rößiger, 2000a) one needs infinitary conjunctions or an infinite set of formulae.

The second line of research that connects modal logic and coalgebra tries to enrich coalgebraic specification with modalities to express certain properties more succinctly. (Jacobs, 1997b) shows that the infinitary[14] modality always can get its semantics via greatest invariants (contained in some predicate). (Rothe, 2000) picks this idea up and describes method wise infinitary modal operators for CCSL (see also Section 4 in (Rothe et al., 2001)). This section describes Rothes method wise modal operators in the formal context of coalgebraic class signatures over a polymorphic type theory.

In the following I consider infinitary versions of the two modal operators $\Box$ and $\Diamond$. For a (syntactic) predicate P the modality $\Box P$ (always P or henceforth P) holds for a state x of a coalgebra c, if P holds for x and all successor states of x, which can be reached via c. So $\Box P$ is the safety property that assures that the bad event $\neg P$ never happens. The

[14] Infinitary means here, that the modal operator applies to all following successor states and not only to the next state. Thus is satisfies the schema 4 (Goldblatt, 1992) and, equivalently, the underlying Kripke structure is transitive.

modality $\Diamond$ (eventually) is the dual of $\Box$, that is $\Diamond P = \neg\Box\neg P$. The formula $\Diamond P$ holds for those states x that have at least one successor state that makes P true. Therefore one can view $\Diamond P$ as the liveness property that holds if the good thing P does eventually happen.[15]

The semantics of the modalities is given by the greatest invariant (contained in some predicate), compare Section 2.6.6 and Section 3.4.6. Because CCSL uses strong invariants the greatest invariant exists for all class signatures over a proper plain ground signature (Proposition 3.4.25). For non-plain ground signatures I assume that the predicate lifting of all type constructors C is monotone in its positive positions:

$$P_i \subseteq Q_i, \qquad \text{implies} \qquad \mathsf{Pred}_{\mathsf{C}}(\top, P_1, \ldots, \top, P_n) \subseteq \mathsf{Pred}_{\mathsf{C}}(\top, Q_1, \ldots, \top, Q_n) \quad (4.1)$$

The interaction of the modalities with the higher-order logic of Definition 4.5.1 is a bit tricky, so let me discuss the type of $\Box$ before I present the definition (the following explanation applies to $\Diamond$ in the same way). From the preceding paragraph it is clear that the expression, to which $\Box$ is applied to, must be a predicate on the state space of the coalgebra. Therefore it must be of type $\mathsf{Self} \Rightarrow \mathsf{bool}$. Assume that P is of type Prop and has a free variable $x : \mathsf{Self}$, then we can form the expression $\Box(\lambda x : \mathsf{Self} \,.\, P)$. This expression is again a predicate on the state space, so we have $\Box(\lambda x : \mathsf{Self} \,.\, P) : \mathsf{Self} \Rightarrow \mathsf{Prop}$.

Note that, different to traditional modal logic, the predicate P can contain additional free variables. In this case both $\Box(\lambda x : \mathsf{Self} \,.\, \forall a : \tau \,.\, P)$ and $\forall a : \tau \,.\, \Box(\lambda x : \mathsf{Self} \,.\, P)$ are possible.

When working with class signatures, one often wants to express that only the subset $\{m_1, \ldots, m_n\}$ of all available methods retains a safety property P. Thereby one explicitly allows that a method $m_0 \notin \{m_1, \ldots, m_n\}$ yields a successor state that violates P. This cannot be expressed with the $\Box$ operator described so far. Similarly, for liveness properties one might want to ensure that a state fulfilling P can be reached by only using a subset of all available methods. One example for this is the property F_{finite} from before, where the empty queue is reached by only applying the method top.

The solution of this problem is to annotate the modal operators with sets of method identifiers, like in $\Box^{\{m_1,m_2\}}(\lambda x : \mathsf{Self} \,.\, P)$. The annotation restricts the number of successor states that are considered: The formula $\Box^M(\lambda x : \mathsf{Self} \,.\, P)$ holds for y if P holds for all successor states that can be reached with methods in M. The value of P on a successor state that was obtained via a method $m \notin M$ does not play any role. Similarly for $\Diamond$.

Definition 4.5.7 (Modal Operators) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature over a proper ground signature. The set of terms over Σ contains in addition to Definition 4.5.1

[15] Note that in general $\Diamond P$ is not a liveness property according to the rigorous definition of (Alpern and Schneider, 1985): $\Diamond P$ is not dense in the obvious topology. However $\Box P$ is closed and therefore a safety property in the sense of Alpern and Schneider.

always

$$\frac{\Xi \mid \Gamma \vdash P : \mathsf{Self} \Rightarrow \mathsf{Prop}}{\Xi \mid \Gamma \vdash \Box^M P : \mathsf{Self} \Rightarrow \mathsf{Prop}}\ M \subseteq \Sigma_M$$

eventually

$$\frac{\Xi \mid \Gamma \vdash P : \mathsf{Self} \Rightarrow \mathsf{Prop}}{\Xi \mid \Gamma \vdash \Diamond^M P : \mathsf{Self} \Rightarrow \mathsf{Prop}}\ M \subseteq \Sigma_M$$

Figure 4.9.: Derivation rules for the modal operators over a class signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$.

- $\Box^M P : \mathsf{Self} \Rightarrow \mathsf{Prop}$ for a set of method (identifiers) $M \subseteq \Sigma_M$ and a predicate $P : \mathsf{Self} \Rightarrow \mathsf{Prop}$.

The derivation rule is in Figure 4.9. There are the following abbreviations

- $\Diamond^M P \stackrel{\text{def}}{=} \lambda y : \mathsf{Self}\,.\, \neg\ \Box^M(\lambda x : \mathsf{Self}\,.\, \neg P\, x)(y)$
- $\Box P \stackrel{\text{def}}{=} \Box^{\Sigma_M} P$
- $\Diamond P \stackrel{\text{def}}{=} \Diamond^{\Sigma_M} P$

Remark 4.5.8 There are several alternatives to introduce modal operators into the higher-order logic of CCSL. The preceding definition seems to be the most appropriate compromise for getting the succinctness of modal logics without cluttering CCSL too much. The following alternatives have been discarded.

- Rößiger and Jacobs define in (Rößiger, 2000a) and (Jacobs, 2000) the notion of paths by induction on the structure of the signature functor. One path denotes precisely one possible way to extract a successor state. Path-wise modal operators allow one to distinguish between several successor states that are obtained from one method application. As an example consider a method (or a coalgebra) $m : \mathsf{Self} \ \Rightarrow\ (\mathsf{Self} \times \mathsf{Self}) + \mathsf{Self}$. For this method there are three paths $\kappa_1 \cdot \pi_1$, $\kappa_1 \cdot \pi_2$, and κ_2. If $m\, x = \kappa_1(y_1, y_2)$ then the path $\kappa_1 \cdot \pi_2$ denotes y_2 and in this case the path κ_2 does *not* denote a successor state.

  Path wise modal operators are finer than method wise ones. However, for CCSL the granularity of methods seems appropriate.

- A modal μ–calculus (Stirling, 1992) for coalgebras would be even more flexible than path wise modal operators. This remains future work, a first discussion is in (Jacobs, 2002).
- The preceding definition introduces $\Box(\lambda x : \mathsf{Self}\,.\, P)$ as a special term. Alternatively one can assume a (higher-order) function $\Box^M : (\mathsf{Self} \Rightarrow \mathsf{Prop}) \Rightarrow (\mathsf{Self} \Rightarrow \mathsf{Prop})$.

- The operator $\Box$ acts as a variable binder. In Definition 4.5.7 this is captured by requiring that P is of type $\mathsf{Self} \Rightarrow \mathsf{bool}$ (the binding is done by lambda abstraction). In a first-order version the bound variable must be explicit, as in the following (where I neglect the annotation with methods):

$$\frac{\Gamma, x : \mathsf{Self} \;\vdash\; P : \mathsf{Prop}}{\Gamma, y : \mathsf{Self} \;\vdash\; (\Box_x P)\; y : \mathsf{Prop}}$$

Here y is a fresh variable and the x in P is bound by $\Box_x$. The substitution rule is as follows:

$$((\Box_x\; P)\; t)\; [s/z] \quad = \quad \begin{cases} (\Box_x\; P)\; (t[s/z]) & \text{for } x = z \\ (\Box_x\; P[s/z])\; (t[s/z]) & \text{for } x \neq z \end{cases}$$

Definition 4.5.9 (Semantics of $\Box$) Let $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ be a coalgebraic class signature over a proper ground signature Ω and let $\mathcal{A} = \langle X, c, a \rangle$ and $\mathcal{M}_\Omega$ be models for Σ and Ω, respectively. Assume that $\mathcal{M}_\Omega$ satisfies the monotonicity requirement of Equation 4.1. Consider the term $\Xi \mid \Gamma \vdash \Box^M P$ with contexts $\Xi = \alpha_1, \ldots, \alpha_n$ and $\Gamma = x_1 : \sigma_1, \ldots, x_k : \sigma_k$. By definition the set of method annotations M forms a subsignature $\Sigma' = \langle M, \emptyset \rangle \leq \Sigma$. Now, fix an interpretation $U_1, \ldots, U_n$ of the type variables and an interpretation X of Self. Let $\overline{x} : \overline{\sigma}$ denote the tuple $x_1 : \llbracket \sigma_1 \rrbracket_{U_1,\ldots,U_n}(X, X),\; \ldots\; , x_k : \llbracket \sigma_k \rrbracket_{U_1,\ldots,U_n}(X, X)$ for the interpretation of the term variables $x_1, \ldots, x_k$. Then

$$\llbracket \Box^M P \rrbracket^{\mathcal{A}} \quad = \quad \lambda \overline{x} : \overline{\sigma} \,.\, \underline{\left(\llbracket P \rrbracket^{\mathcal{A}} (\overline{x})\right)}_{\pi_M \,\circ\, c}$$

where $\underline{(-)}_{\pi_M \circ c}$ denotes the greatest invariant with respect to the induced coalgebra

$$\pi_{\Sigma'} \circ c : X \longrightarrow \llbracket \tau_{\Sigma'} \rrbracket(X, X)$$

Example 4.5.10 A queue contains a finite number of elements if the top method eventually returns bot. This property can now be formalised as

$$F_{\mathsf{finite}}(q) \quad \stackrel{\mathrm{def}}{=} \quad (\Diamond^{\mathsf{top}}\; \lambda x : \mathsf{Self} \,.\, \mathsf{top}(x) = \mathsf{bot})\; (q)$$

as negation we obtain

$$F_{\mathsf{infinite}}(q) \quad \stackrel{\mathrm{def}}{=} \quad (\Box^{\mathsf{top}}\; \lambda x : \mathsf{Self} \,.\, \neg\; \mathsf{top}(x) = \mathsf{bot})\; (q)$$

For another example recall the formulae F_{empty} and F_{filled} from Example 4.5.2 that capture the behaviour of FIFO queues. Define a formula that describes finite FIFO queues as

$$F_{\mathrm{FFIFO}}(x) \quad \stackrel{\mathrm{def}}{=} \quad F_{\mathsf{empty}}(x) \wedge F_{\mathsf{filled}}(x) \wedge F_{\mathsf{finite}}(x)$$

Let q be an arbitrary finite FIFO queue. Starting from any other finite FIFO queue p one can construct a queue p' such that p' and q are bisimilar. This is expressed with

$$F_{\text{FFIFO}}(p) \wedge F_{\text{FFIFO}}(q) \quad \supset \quad (\Diamond\ \lambda p' : \mathsf{Self}\,.\, p' \sim q)(p) \tag{4.2}$$

with method wise modal operators it can be slightly strengthened to

$$F_{\text{FFIFO}}(p) \wedge F_{\text{FFIFO}}(q) \quad \supset \quad (\Diamond^{\mathsf{top}}\ \Diamond^{\mathsf{put}}\ \lambda p' : \mathsf{Self}\,.\, p' \sim q)(p) \tag{4.3}$$

which expresses that the successor state p' that is equivalent to q can be reached by first emptying p and then filling it. The preceding two statements can be put as theorems into the queue specification, see Figure 4.11 (on page 184 below). Although the proof of 4.2 and 4.3 is intuitively very simple it requires a fair amount of work to prove the two statements in PVS. The proof distributed with the sources of the queue example (see Appendix A) requires 54 utility lemmas that have been proved with about 700 PVS proof commands. ■

In the remainder I show a few general results about the modal operators. I first investigate behavioural invariance, extending Proposition 4.5.6.

Proposition 4.5.11 *Let Σ be a coalgebraic class signature over a plain ground signature that contains only polynomial methods. If the term $\Xi \mid \Gamma \vdash P$ is invariant with respect to behavioural equality, then so is $\Xi \mid \Gamma \vdash \Box^M P$.*

The preceding proposition does not hold if Σ contains a binary method. It is easy to construct an example that shows this. The problem here is that CCSL uses strong invariants and that Proposition 3.4.11 (on page 99) fails for strong invariants.

Proof Under the assumptions of the proposition Σ corresponds to a polynomial functor. Consider two models $\mathcal{A} = \langle X, c, a\rangle$ and $\mathcal{B} = \langle Y, d, b\rangle$ of Σ and let R be a bisimulation for c and d that relates $x \in X$ and $y \in Y$. Assume $x \in [\![\Box^M P]\!]^{\mathcal{A}}$, it remains to show that also $y \in [\![\Box^M P]\!]^{\mathcal{B}}$. Note that R is also a bisimulation for $\pi_M \circ c$ and $\pi_M \circ d$. Consider now the predicate $Q = \coprod_{\pi_2} (R \wedge \pi_1^* \, [\![\Box^M P]\!]^{\mathcal{A}}) = \{y \mid \exists x \,.\, x\, R\, y \wedge x \in [\![\Box^M P]\!]^{\mathcal{A}}\}$. By Proposition 2.6.17 and 2.6.15 Q is an invariant for $\pi_M \circ d$. And because P is behaviourally invariant, we have $Q \subseteq [\![P]\!]^{\mathcal{B}}$. □

Proposition 4.5.12 ((Rothe, 2000)) *The method wise modal operators $\Box^M$ fulfil the* S4 *rules. For arbitrary predicates $\Xi \mid \Gamma \vdash P : \mathsf{Self} \Rightarrow \mathsf{bool}$ and $\Xi \mid \Gamma \vdash Q : \mathsf{Self} \Rightarrow \mathsf{bool}$ with $x \notin \Gamma$ we have*

$$\begin{array}{rlll} K: & \forall x : \mathsf{Self}\,. & \big(\Box^M\ \lambda x : \mathsf{Self}\,.\,(P\,x \supset Q\,x)\big)(x) & \supset \quad (\Box^M\ P)(x) \ \supset\ (\Box^M\ Q)(x) \\ T: & \forall x : \mathsf{Self}\,. & (\Box^M P)(x) \quad \supset \quad P(x) & \\ 4: & \forall x : \mathsf{Self}\,. & (\Box^M P)(x) \quad \supset \quad (\Box^M\ \Box^M P)(x) & \end{array}$$

As usual in modal logic one can get theorems for $\Diamond^M$ by dualization, for instance, the dualized version of T is

$$\forall x : \mathsf{Self}\,.\quad P(x) \quad \supset \quad (\Diamond^M\ P)(x)$$

Proof The proof follows from basic properties of greatest invariants, for instance $[\![\Box^M\ P]\!]$ is an invariant, therefore $[\![\Box^M\ \Box^M\ P]\!] = [\![\Box^M\ P]\!]$, which implies 4. □

Proposition 4.5.13 *Let* $\Xi \mid \Gamma, y : \tau \vdash P : \mathsf{Self} \Rightarrow \mathsf{bool}$ *be a predicate, which possibly contains* y *freely. Then for all* $x : \mathsf{Self}$ *it holds that*

1. $$\forall y : \tau\,.\,(\Box^M\ P)(x) \quad \supset\!\!\!\subset \quad \big(\Box^M\ \lambda x' : \mathsf{Self}\,.\,\forall y : \tau\,.\,P(x')\big)(x)$$
2. $$\exists y : \tau\,.\,(\Box^M\ P)(x) \quad \supset \quad \big(\Box^M\ \lambda x' : \mathsf{Self}\,.\,\exists y : \tau\,.\,P(x')\big)(x)$$

Proof For 1 it suffices to prove that $Q(x) \overset{\text{def}}{=} \forall y : \tau\,.\,(\Box^M\ P)(x)$ is the greatest invariant implying $\forall y : \tau\,.\,P(x)$. Clearly, the predicate Q is an invariant that implies $\forall y : \tau\,.\,P(x)$. Now assume an invariant Q' such that $Q'(x)$ implies $\forall y : \tau\,.\,P(x)$. Then $Q'(x)$ implies also $P(x)$ for an arbitrary but fixed y. By definition $\Box^M\ P(x)$ is the greatest invariant implying $P(x)$ for any fixed y, therefore $Q'(x)$ implies also $\Box^M\ P(x)$ for any fixed y. Because the y was chosen arbitrarily, $Q'(x)$ implies $\forall y : \tau\,.\,\Box^M\ P(x)$, so Q is indeed the greatest invariant.

For 2 it suffices to show that $\exists y : \tau\,.\,(\Box^M\ P)$ is an invariant that implies $\exists y : \tau\,.\,P$. Both is obvious. □

4.5.3. Coalgebraic Class Specifications

A specification is a signature whose class of models is restricted with a set of axioms.

Definition 4.5.14 (Coalgebraic Class Specification) Let Σ be a coalgebraic class signature with type parameters $\Xi = \alpha_1, \ldots, \alpha_n$.

1. A formula φ is a Σ *method assertion*, if $\Xi \mid x : \mathsf{Self} \vdash \varphi$ and if φ contains no constructor from Σ_C.
2. A formula ψ is a Σ *constructor assertion* if $\Xi \mid \emptyset \vdash \psi$.
3. A *coalgebraic class specification* is a triple $\langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$ where Σ is a coalgebraic class signature, $\mathcal{A}_M$ is a finite set of Σ method assertions, and $\mathcal{A}_C$ is a finite set of Σ constructor assertions.

Example 4.5.15 In a class specification for queues it makes sense to demand that the queue constructor new returns an empty queue. Therefore I set

$$\mathsf{Queue} = \langle \Sigma_{\mathsf{Queue}},\ \{F_{\mathsf{empty}}, F_{\mathsf{filled}}\},\ \{F_{\mathsf{new}}\} \rangle$$

where

$$F_{\mathsf{new}} \quad = \quad \left[\quad \mathsf{top(new)} \; = \; \mathsf{bot} \quad \right]$$

specifies that new delivers the empty queue. ■

The notion of a *subspecification* is needed below. The restriction to method assertions is implied by the restriction to method declarations in subsignatures and will be explained in Subsection 4.8.1 (on page 210) below.

Definition 4.5.16 (Subspecification) A class specification $\mathcal{S}' = \langle \Sigma', \mathcal{A}'_M, \mathcal{A}'_C \rangle$ is a *subspecification* of $\mathcal{S} = \langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$, denoted as $\mathcal{S}' \leq \mathcal{S}$ if $\Sigma' \leq \Sigma$ and $\mathcal{A}'_M \subseteq \mathcal{A}_M$.

Definition 4.5.17 (Semantics of Class Specifications) Let $\langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$ be a coalgebraic class specification. A *model* of this class specification is a model $\mathcal{M}$ of Σ such that for all interpretations U_i of the type variables the following holds.

- For all $x \in X$ all method assertions hold: $[\![\varphi]\!]^{\mathcal{M}}\, x = \top$ for all $\varphi \in \mathcal{A}_M$.
- All constructor assertions are fulfilled: $[\![\psi]\!]^{\mathcal{M}} = \top$ for $\psi \in \mathcal{A}_C$.

Example 4.4.6 has been carefully constructed, it actually is a model of the Queue–specification from Example 4.5.15. A class specification is *consistent* if it has at least one model with a nonempty state space. Note that for a consistent class specification the models form always a proper class.

Assume a subspecification $\mathcal{S}'$ of $\mathcal{S}$ (involving $\Sigma' \leq \Sigma$) and a model $\mathcal{M} = \langle X, c, a \rangle$ of $\mathcal{S}$. The coalgebra c fulfils all assertions of $\mathcal{S}$, so it obviously fulfils the assertions of $\mathcal{S}'$. Therefore also $\pi_{\Sigma'} \circ c$ fulfils all assertions of $\mathcal{S}'$.

The following standard result describes under which condition one can obtain a final coalgebra satisfying the method assertions of a specification. A first version of it appeared in (Jacobs, 1996b).

Proposition 4.5.18 *Let $\mathcal{S} = \langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$ be a consistent coalgebraic class specification over a plain ground signature which contains only polynomial methods. Let τ_Σ be its combined method type. If all method assertions and constructor assertions of $\mathcal{S}$ are invariant with respect to behavioural equality, then there exists a model $\mathcal{M} = \langle X, c, a \rangle$ of $\mathcal{S}$ such that c is the final τ_Σ coalgebra satisfying the method assertions $\mathcal{A}_M$.*

Proof Under the assumptions the semantics of τ_Σ is a polynomial functor. For polynomial functors coalgebra morphisms are bisimulations (Proposition 2.6.12), therefore every τ_Σ coalgebra morphism preserves the validity of the method and constructor assertions. Let $z : Z \longrightarrow [\![\tau_\Sigma]\!](Z)$ be the final τ_Σ coalgebra (which exists by Theorem 2.6.20). Let X be the greatest invariant contained in the interpretation of the method assertions on Z. By Proposition 2.6.8 there is an induced coalgebra structure on X, this gives c. It remains to construct the constructor algebra for X. Note that X is nonempty because it must contain the image of the state space of the assumed model. Therefore one can set $a = \,!\, \circ a'$, where a' is a constructor algebra of an arbitrary model. □

4.5.4. Class Specifications in CCSL

The concrete syntax of the higher-order logic for CCSL is in Figure 4.10. CCSL follows quite closely the concrete syntax of PVS. Especially the ASCII representations of the logical notation and the projections is the same as in PVS. There are the following points to note with respect to Figure 4.10:

- CCSL has concrete syntax for expressions that are syntactic sugar with respect to Definition 4.5.1, for instance quantification and abstraction over several variables simultaneously, the `LET` construct, function update with `WITH`, or the keyword `IFF`.

- The keywords `ALWAYS` and `EVENTUALLY` give the modal operators from the preceding subsection. The correspondence between the symbolic notation of this chapter and the concrete grammar of CCSL is as follows:

$$\Box^{M}\,(\lambda x : \mathsf{Self}\,.\,P) \quad\equiv\quad \texttt{ALWAYS LAMBDA(}\ x : \texttt{SELF)}\ .\ P\ \texttt{FOR}\ \{\ M\ \}$$
$$\Diamond^{M}\,(\lambda x : \mathsf{Self}\,.\,P) \quad\equiv\quad \texttt{EVENTUALLY LAMBDA(}\ x : \texttt{SELF)}\ .\ P\ \texttt{FOR}\ \{\ M\ \}$$

  The optional identifier with an argument list before the method list can be used to access a modal operator of a different class specification. If it is omitted it defaults to the enclosing class specification.

- The syntax for `CASES` provides terms for the abstract data types. Abstract data types are in Section 4.6. I define the semantics of the case construct in Subsection 4.7.1. The case construct in *Terms*(Σ) corresponds to case construct in CCSL for the abstract data type Coproduct, which is defined in the prelude (see Example 4.3.2 and Subsection 4.9.8).

- CCSL allows object-oriented syntax for method calls: One can write $x.m(-)$ instead of $m(x,-)$ to give the specifications a bit more object-oriented look and feel.

- The projections are `PROJ_N`, where `N` stands for a natural number.

- Infix operators are sequences of special characters like `*`, the details are in Subsection 4.9.5.

- CCSL tries to be relaxed about the use of delimiters. For instance, variable binders like `FORALL` can be separated from the following formula by either a colon or a dot (thus comforting users of PVS *and* ISABELLE). The last delimiter in a list of cases or let bindings is optional.

- The precedence for the constructions in Figure 4.10 increases from the top to the bottom. So conjunction (`AND`) binds stronger then disjunction (`OR`). For instance the expression f_1 `OR` $f_2\ f_3\,.\,\mathsf{m}$ is parsed as f_1 `OR` $((f_2\ f_3)\,.\,\mathsf{m})$.

```
formula     ::= FORALL ( vardecl ⦃ , vardecl ⦄ ) ( : | . ) formula
              | EXISTS ( vardecl ⦃ , vardecl ⦄ ) ( : | . ) formula
              | LAMBDA ( vardecl ⦃ , vardecl ⦄ ) ( : | . ) formula
              | LET binding ⦃ ( ; | , ) binding ⦄ [ ; | , ] IN formula
              | formula IFF formula
              | formula IMPLIES formula
              | formula OR formula
              | formula AND formula
              | IF formula THEN formula ELSE formula
              | NOT formula
              | formula infix_operator formula
              | ALWAYS formula FOR
                [ identifier [ argumentlist ] : : ] methodlist
              | EVENTUALLY formula FOR
                [ identifier [ argumentlist ] : : ] methodlist
              | CASES formula OF caselist [ ; | , ] ENDCASES
              | formula WITH [ update ⦃ , update ⦄ ]
              | formula . qualifiedid
              | formula formula
              | TRUE
              | FALSE
              | PROJ_N
              | number
              | qualifiedid
              | ( formula : type )
              | ( formula ⦃ , formula ⦄ )
vardecl     ::= identifier ⦃ , identifier ⦄ : type
methodlist  ::= { identifier ⦃ , identifier ⦄ }
binding     ::= identifier [ : type ] = formula
caselist    ::= pattern : formula ⦃ ( ; | , ) pattern : formula ⦄
pattern     ::= identifier [ ( identifier ⦃ , identifier ⦄ ) ]
update      ::= formula := formula
```

Figure 4.10.: CCSL Syntax for expressions and formulae

- The CCSL compiler defines four special class members that make the notions of invariants, bisimulations, coalgebra morphisms (in the form of recognisers), and coinduction (in the form of the coreduce combinator) available in the logic of CCSL. Their names are as follows:

concept	identifier for class ⟨`class`⟩	relevant definition
invariant	⟨`class`⟩`_class_invariant?`	Definition 4.4.10 (1)
bisimulation	⟨`class`⟩`_class_bisimulation?`	Definition 4.4.10 (3)
morphism	⟨`class`⟩`_class_morphism?`	Definition 3.2.2
coinduction	`coreduce`	Item Coreduce on page 201

  The types of these identifiers depend on the method declarations that are present in the signature, see Subsection 4.7.1 for the details. The identifier for coreduce does not depend on the class name, so one usually has to use a qualified identifier for it (see Subsection 4.9.6 on page 222). Figure 4.11 shows as an example how to express in CCSL that the predicate F_{finite} from Example 4.5.10 is an invariant for queues.

  Apart from `coreduce` (whose semantics also depends on the method assertions) these identifiers are visible in method and constructor assertions. (Technically the compiler makes a ground signature extension just before processing method and constructor assertions.)

- The current compiler version supports only one kind of immediate constants: natural numbers. Their type can be changed via the `-nattype` command line switch, see Subsection 4.9.9.

The syntax for class specifications was already given in Subsection 4.4.2. Recall that the sections for attributes, methods and constructors contributed to the signature. The section for assertions contains method assertions in the sense of Definition 4.5.14, the section for creation conditions contains constructor assertions. The theorem section gives one the opportunity to state theorems in the logic of CCSL that are believed to hold for all models. From the point of view of the CCSL compiler the theorem section contains arbitrary formulae (without influence on the semantics of the class specification) that should get translated into the logic of the target theorem prover. The syntax of these sections is as follows:

$$\begin{array}{lcl}
\textit{assertionsection} & ::= & \texttt{ASSERTION}\ \{\!|\ \textit{importing}\ |\!\}\ [\ \textit{assertionselfvar}\ ] \\
 & & \{\!|\ \textit{freevarlist}\ |\!\}\ \textit{namedformula}\ \{\!|\ \textit{namedformula}\ |\!\} \\
\textit{assertionselfvar} & ::= & \texttt{SELFVAR}\ \textit{identifier}\ \texttt{:}\ \texttt{SELF} \\
\textit{freevarlist} & ::= & \texttt{VAR}\ \textit{vardecl}\ \{\!|\ \texttt{;}\ \textit{vardecl}\ |\!\} \\
\textit{creationsection} & ::= & \texttt{CREATION}\ \{\!|\ \textit{importing}\ |\!\}\ \{\!|\ \textit{freevarlist}\ |\!\} \\
 & & \textit{namedformula}\ \{\!|\ \textit{namedformula}\ |\!\}
\end{array}$$

theoremsection ::= `THEOREM` ⦃ *importing* ⦄ ⦃ *freevarlist* ⦄
namedformula ⦃ *namedformula* ⦄

namedformula ::= *identifier* : *formula* ;

All three sections can contain an arbitrary number of (named) formulae. The importings are explained in Section 4.9.4. Every assertion section can contain a `SELFVAR` declaration for the free variable that can occur in method assertions. Variables declared with the keyword `VAR` constitute a context for all formulae of the affected section. This is syntactic sugar: The CCSL compiler universally quantifies all the variables declared with `VAR` on the outermost level.

The complete queue specification in CCSL syntax is in Figure 4.11. Besides the three assertions F_{empty}, F_{filled}, and F_{new} from Example 4.5.15 it contains also two theorems. The first one corresponds to Equation 4.3 and the second one states that the predicate F_{finite} from Example 4.5.10 is an invariant.[16] The CCSL compiler translates the two theorems into lemmas in a separate PVS file. The utility lemmas that are necessary for the two theorems are in the PVS theory QueueModal, to make them available I add an appropriate importing clause.

In the remainder of this subsection I show how the CCSL compiler translates the two queue assertions into PVS. Further below I discuss how the compiler treats attribute declarations and how the associated update assertions look like.

During type checking the CCSL compiler records that the queue assertions use behavioural equality on the type Lift[A × Self]. Therefore it generates the theory QueueReqObsEq, which contains the following lifting of bisimilarity to the type Lift[A × Self].

```
c : Var QueueSignature[Self, A]

ObsEq_Lift_A_Self(c) : [[Lift[[A, Self]], Lift[[A, Self]]] -> bool] =
    Lambda(l1 : Lift[[A, Self]], l2 : Lift[[A, Self]]) :
        Cases l1 OF
            bot    : bot?(l2),
            up(p0) : up?(l2)    And   PROJ_1(p0) = Proj_1(down(l2))   And
                          bisim?(c)(PROJ_2(p0), PROJ_2(down(l2)))
        Endcases
```

The method assertions are translated into predicates on queue coalgebras. For the method assertion q_empty the compiler generates the predicate q_empty?. The follow-

[16] The distributed sources contain also a theorem that corresponds to Equation 4.2. Unfortunately it does not fit into the figure.

```
Begin Queue[ A : Type ] : ClassSpec
  Method
    put : [Self, A] -> Self;
    top : Self -> Lift[[A,Self]];

  Constructor
    new : Self;

  Assertion Selfvar x : Self
    q_empty :  x.top ~ bot      Implies
        Forall(a : A) . x.put(a).top ~ up(a,x);

    q_filled :   Forall(a1 : A, y : Self) . x.top ~ up(a1, y)      Implies
        Forall(a2 : A) . x.put(a2).top ~ up(a1, y.put(a2));

  Creation
    q_new : new.top ~ bot;

  Theorem
    Importing QueueModal[Self, A]

    strong_reachable :  Forall(p, q : Self) :
      Let finite? : [Self -> bool] = Lambda(q : Self) :
              (Eventually Lambda(x : Self) : x.top = bot For {top}) q
      IN
        finite? p And finite? q   Implies
          (Eventually
                (Eventually Lambda(r : Self) : r ~ q For {put})
           For {top}
          ) p;

    finite_invariant :
      Let finite? : [Self -> bool] = Lambda(q : Self) :
              (Eventually Lambda(x : Self) : x.top = bot For {top}) q
      IN
        Queue_class_invariant?(put, top)(finite?) ;
End Queue
```

Figure 4.11.: The queue specification in CCSL syntax

ing material is taken from the theory QueueSemantics.

```
q_empty?(c) : [Self -> bool] = Lambda (x : Self) :
    ObsEq_Lift_A_Self(c)(top(c)(x), bot)      Implies
    Forall(a : A) : ObsEq_Lift_A_Self(c)(top(c)(put(c)(x, a)), up(a, x))
```

Observe that the compiler inserts coalgebras to make the methods dependent on a model. The assertion q_filled is translated into PVS in the same way. All such translated method assertions are combined into one ⟨class⟩Assert? predicate, which holds precisely on those coalgebras that fulfil all method assertions.

```
QueueAssert?(c) : bool =
    Forall(x : Self) : q_empty?(c)(x)   And  q_filled?(c)(x)
```

In a similar way the constructor assertions are translated into predicates on the constructor algebra. Like in the queue example the constructor assertions can contain methods. Therefore the predicates for the constructor assertions depend on an interpretation of the methods. In the PVS translation this is captured with an additional argument c:

```
q_new?(c) : [QueueConstructors[Self, A] -> bool] =
    Lambda(z : QueueConstructors[Self, A]) :
        ObsEq_Lift_A_Self(c)(top(c)(new(z)), bot)

QueueCreate?(c) : [QueueConstructors[Self, A] -> bool] =
    Lambda(z : QueueConstructors[Self, A]) : q_new?(c)(z)
```

Finally the predicates ⟨class⟩Assert and ⟨class⟩Create are combined into an recogniser of queue models:

```
QueueModel?(c : QueueSignature[Self, A], z : QueueConstructors[Self, A]) : bool=
    QueueAssert?(c)   And   QueueCreate?(c)(z)
```

The construction of a model of the queue specification in the target theorem prover consists of a prove of a theorem of the form

```
model : Proposition QueueModel?[QueueState, A](Queue_c, Queue_constr)
```

where QueueState is the type of the state space of the model and A is a type parameter. The records Queue_c and Queue_constr contain (the user defined) interpretation of the queue signature.

As I said before an important task of the CCSL compiler is the generation of lemmas. For every method and constructor assertion the compiler generates one lemma. The lemma for the assertion q_empty looks as follows:

```
q_empty : Lemma
    Forall(c : QueueSignature[Self, A], x : Self) : QueueAssert?(c)      Implies
        ObsEq_Lift_A_Self(c)(top(c)(x), bot)      Implies
            Forall(a : A) : ObsEq_Lift_A_Self(c)(top(c)(put(c)(x, a)), up(a, x))
```

Let me now discuss attribute declarations and their associated assertions. If the signature of the class contains attributes, then the compiler generates not only additional method declarations. It also generates additional assertions that describe the behaviour of the attributes with respect to the generated update methods. Assume for an example a specification with the following attribute declarations (where U and V are type parameters):

```
Attribute
  a1 : Self -> Bool;
  a2 : [Self, U] -> V;
```

As update methods the compiler generates the following two method declarations.

```
  set_a1 : [Self, Bool] -> Self;
  set_a2 : [Self, U, V] -> Self;
```

For each combination of attribute and update method there is an *update assertion* generated that describes if and how the attribute changes. In CCSL syntax these assertions would look as follows.

```
Assertion SelfVar x : Self
  a1_set_a1 : Forall (y : Bool) : a1(set_a1(x, y)) = y;

  a1_set_a2 : Forall (u : U, v : V) : a1(set_a2(x, u, v)) = a1(x);

  a2_set_a1 : Forall (y : Bool, u : U) : a2(set_a1(x, y), u) = a2(x, u);

  a2_set_a2 : Forall (u1 : U, u2 : U, v : V) : a2(set_a2(x, u1, v), u2) =
                              IF u1 = u2 Then v Else a2(x, u2);
```

4.6. Abstract Data Types

Abstract data types are widely accepted as the right formalism to specify finitely generated data structures such as lists or trees. Many functional programming languages (e.g., SML (Milner et al., 1991), OCAML (Leroy et al., 2001), and Haskell (Augustsson et al., 1999; Hudak et al., 1992)) allow the definition of abstract data types. The logic of the theorem provers PVS and ISABELLE/HOL has been extended with means to specify abstract data types (Owre and Shankar, 1993; Berghofer and Wenzel, 1999).

Although it is possible, it does not make much sense to model abstract data types with behavioural types. Therefore CCSL contains the possibility to specify abstract data types as initial algebras. This way, the decision whether to choose an algebraic or a coalgebraic approach to model a given type is left to the user. Further, it is possible to mix abstract data type specifications with coalgebraic class specifications, this leads to iterated specifications, see Section 4.7.

For reasons that have been described in the introduction of this chapter the CCSL compiler accepts currently only abstract data type specifications without axioms. As a consequence also this chapter restricts to abstract data types. There is no problem with general algebraic specifications. The extension of CCSL with general algebraic specifications is one of the points that remain to be done in the future.

Definition 4.6.1 (ADT) Assume a ground signature Ω. An *abstract data type specification* is a finite set Σ of constructor declarations $c_i : \sigma_i$ where σ is a constructor type. The type variables occurring in the σ_i are the *type parameters* of the abstract data type specification.

Recall from Definition 4.2.7 (on page 142) that a constructor type is a type expression $\sigma \Rightarrow \mathsf{Self}$ such that Self occurs in σ only strictly positive. This restriction in the above definition is necessary, because initial algebras exist only for certain functors (Gunter, 1992; Owre and Shankar, 1993; Berghofer and Wenzel, 1999).

The semantics of abstract data type specifications is given by a collection of initial algebras.

Definition 4.6.2 Let Σ be an abstract data type specification with n type parameters and k constructors $c_1, \ldots, c_k$. Let $\sigma_\Sigma = \mathcal{T}_C(c_1) + \cdots + \mathcal{T}_C(c_k)$ denote the combined constructor type of Σ. A *model* for Σ is an indexed collection of pairs $\big(\langle X, a\rangle_{U_1,\ldots U_n}\big)_{U_i \in |\mathbf{Set}|}$ such that for each interpretation $U_1, \ldots, U_n$ of the type variables

$$[\![\sigma_\Sigma]\!]_{U_1,U_1,U_2,U_2,\ldots,U_n,U_n}(X, X) \xrightarrow{\quad a \quad} X$$

is an initial algebra.

4.6.1. CCSL Syntax for Abstract Data Types

The concrete syntax for abstract data type specifications is similar to that of class specifications. The keyword `ADT` indicates an abstract data type.

adtspec	::=	`BEGIN` *identifier* [*parameterlist*] `: ADT` ⦃ *adtsection* ⦄ `END` *identifier*

```
Begin tree[A, B : Type] : Adt
   Constructor
      leaf : B -> Carrier;
      node : [Carrier, A, Carrier] -> Carrier
End tree
```

Figure 4.12.: The abstract data type of binary trees in CCSL

```
adtsection           ::=  adtconstructorlist [ ; ]
adtconstructorlist   ::=  CONSTRUCTOR adtconstructor ⦃ ; adtconstructor ⦄
adtconstructor       ::=  identifier [ adtaccessors ] : type
                       |  identifier [ adtaccessors ] : type -> type
adtaccessors         ::=  ( identifier ⦃ , identifier ⦄ )
```

The accessors are syntactic sugar, so let me ignore them for a moment. The set of declared adt-constructors constitute an abstract data type specification. The compiler checks that the types are constructor types. Recall, that for this presentation I assume only one special type Self, whereas the CCSL compiler has two keywords for it, `SELF` and `CARRIER`. In abstract data type specification one has to use the latter one.

From every constructor the compiler derives a recogniser predicate by appending a question mark (for PVS) or prepending the prefix is_ (for ISABELLE). A recogniser holds for an element of the abstract data type if this element was built with the corresponding constructor.

The optional accessors declare (partial) accessor function. If accessors are given then their number must match the number of arguments of the constructor and their names must be unique. Accessors allow one to decompose an element of the abstract data type and extract the arguments of the constructor with which this element was built. Because accessors are partial functions, they cannot be formalised in the setting of this thesis. Nevertheless the CCSL compiler allows them. The compiler together with the semantics of the theorem provers PVS and ISABELLE/HOL ensure a correct treatment of these partial functions (I discussed this issue already in Example 4.3.2).

Figure 4.12 shows as an example the abstract data type of binary trees in CCSL syntax.

The compiler does not generate the semantics for abstract data types. It rather outputs an abstract data type declaration in the syntax of the target theorem prover. Both ISABELLE/HOL and PVS use an initial semantics for their abstract data types. In ISABELLE this is implemented as a conservative extension[17] (Berghofer and Wenzel, 1999); PVS uses an axiomatic approach (Owre and Shankar, 1993).

[17] Provided the `quick_and_dirty` flag is set to false.

The simple mapping of CCSL data type definitions to the target theorem prover has one serious drawback: Inside an abstract data type all type constructors stemming from a class specification may only be instantiated with constant types. This is because both PVS and ISABELLE place restrictions on the types that may be used in a nested recursion (on the type level) with an abstract data type definition. In principle, nested recursion with (some) behavioural types could be allowed, see the following Section 4.7.

The theorem provers PVS and ISABELLE give different support for their versions of abstract data types: Recognisers, accessors, and the map combinator are not provided by the data type package of ISABELLE/HOL. More importantly, some notions, which are needed when abstract data types occur inside coalgebraic class specifications, are not supported in the needed generality or are not supported at all. For instance neither PVS nor ISABELLE defines relation lifting for abstract data types. PVS generates for every abstract data type the combinators every and map. For an abstract data type in which all type variables occur at strictly positive position the combinator every coincides with predicate lifting and the combinator map with the morphism component of the semantics of the data type. If a type variable occurs not strictly positive then PVS does not generate map. The combinator every is still generated but disregards all type variables occurring not only strictly positive. So in this case every cannot be used for predicate lifting.

The CCSL compiler works hard to blur the differences between data type definitions in PVS and ISABELLE. It also fixes some of their shortcomings. For ISABELLE the compiler generates definitions for recogniser predicates, and accessor functions. For both PVS and ISABELLE the compiler generates predicate and relation lifting for the abstract data type as described in (Hensel, 1999) and in the following Section.

In the remainder of this section I show what the CCSL compiler generates for the data type of trees of Figure 4.12. For a diversion I show this time what is generated for ISABELLE/HOL.

As first the data type tree is defined:

```
datatype ('A, 'B) tree =
    leaf "'B"
  | node "('A, 'B) tree"   "'A"   "('A, 'B) tree"
```

ISABELLE/HOL data type declarations have an SML like syntax. Different constructors are separated with a vertical bar and the argument types follow the name of the constructor. Identifiers that start with a tick like 'A are free type variables. Instantiations of type constructors are written in a postfix form. The ISABELLE type expression ('A, 'B) tree corresponds to tree[A, B] in PVS. ISABELLE requires the user to enclose all special syntax from the object logic in double quotes. The CCSL compiler behaves conservatively and puts double quotes around all critical entities.

For the definition of functions over data types the ISABELLE/HOL documentation advocates the **primrec** feature. However, I found that functions defined with **primrec** are quite slow to type check. More importantly, **primec** is quite difficult to use with nested data types.[18] I therefore decided to base all definitions on a self-defined reduce combinator. This has the additional advantage that more modules in the CCSL compiler get independent of the target theorem prover (in PVS the reduce combinator is provided by the system). The disadvantage is that for ISABELLE the CCSL compiler has to derive the reduce combinator from the internal recursion combinator of ISABELLE. In the tree example this looks as follows:

```
constdefs reduce_tree :: "('B => 'Result) =>
                        ('Result => 'A => 'Result => 'Result) =>
                        ('A, 'B) tree => 'Result"
  "reduce_tree leaf_fun node_fun   ==   %(t :: ('A, 'B) tree) .
     tree_rec ( %(b1 :: 'B) .  leaf_fun b1)
              ( %(x1 :: ('A, 'B) tree)  (a1 :: 'A)   (x2 :: ('A, 'B) tree)
                   (x3 :: 'Result)   (x4 :: 'Result) . node_fun x3 a1 x4) t"
```

In ISABELLE a constant definition starts with the keyword **constdefs**, followed by a type annotation (on the first three lines) and a string that contains a meta equality (==). The higher-order function reduce_tree takes as argument a tree algebra on 'Result (consisting of two functions, one for the constructor leaf and one for node). It returns the unique tree–algebra morphism originating in the initial tree algebra (compare Item reduce in Subsection 4.7.1 on page 195). The definition of reduce_tree uses ISABELLE's internal recursion operator tree_rec. This internal combinator could be paraphrased as strong reduce combinator for the unfolded data type. The percent sign % is the ASCII version of λ in ISABELLE.

The reduce combinator gives the induction proof principle (sometimes known as primitive recursion) for trees. It allows one to define recursive functions on trees without doing recursion. For instance to count the number of nodes in a tree t one can use the following expression:

```
reduce_tree (% (b :: 'B) . 0)   (% (l :: nat) (a :: 'A) (r :: % nat) . l + r +1)  t
```

The data type package of ISABELLE/HOL does neither provide accessors nor recogniser predicates. The CCSL compiler generates code to define them. For instance:

```
constdefs leaf_acc :: "('A, 'B) tree => 'B"
  "leaf_acc t == case t of
       leaf b => b
     | node p0 a p1 => arbitrary"
```

In case one applies leaf_acc to a node one gets arbitrary as result, where arbitrary is special constant that inhabits every type.[19]

[18] The problem is that one cannot pass the function being defined by the current **primrec** into an already defined (higher-order) function. Therefore, for nested data types, one cannot use the map combinator of the nested data type.

[19] Recall that the semantics of ISABELLE/HOL is based on an universe of nonempty sets. So arbitrary can be obtained by applying the Axiom of Choice to every type.

The recognisers can be easily defined with reduce, as example I show the recogniser for leafs:

```
constdefs is_leaf :: "('A, 'B) tree => bool"
  "is_leaf == % (t :: ('A, 'B) tree) .
     reduce_tree (% (b :: 'B) . True)
                 (% (x1 :: bool) (a :: 'A) (x2 :: bool) . False) t"
```

There are three more definitions that are generated for every abstract data type: The map combinator, predicate lifting and relation lifting. Here I show only the map combinator and predicate lifting. Relation lifting for abstract data types is quite difficult to understand. It is explained in Item 3 of Remark 4.7.1 (on page 197f).

```
constdefs treeMap :: "('A1 => 'A2) => ('B1 => 'B2) =>
                      ('A1, 'B1) tree => ('A2, 'B2) tree"
  "treeMap f g == % (t :: ('A1, 'B1) tree) .
     reduce_tree (% (b :: 'B1) . leaf (g b))
                 (% (p0 :: ('A2, 'B2) tree) (a :: 'A1)
                 (p1 :: ('A2, 'B2) tree) . node p0 (f a) p1) t"
```

The map combinator takes two functions as arguments, one for transforming the labels in the leafs and one for the labels of the nodes and applies both functions recursively in the whole tree.

```
constdefs
  Everytree :: "('A => bool) => ('B => bool) => ('A, 'B) tree => bool"
  "Everytree P Q ==
       reduce_tree (% (b :: 'B) . Q b)
                   (% (x1 :: bool) (a :: 'A)  (x2 :: bool) .
                         (x1 = True) & (P a) & (x2 = True))"
```

Predicate lifting takes two argument predicates, one on the type parameter A and one on B. It applies these predicates in the whole tree and returns true if all labels are in the supplied predicates (the ampersand & denotes conjunction in ISABELLE).

4.7. Iterated Specifications

In analogy with the *iterated data types* of (Hensel, 1999) and (Hensel and Jacobs, 1997) I use the informal term *iterated specification* to describe a situation in which a specification $\mathcal{S}_i$ depends on a specification $\mathcal{S}_j$. Both involved specifications $\mathcal{S}_i$ and $\mathcal{S}_j$ can be either coalgebraic class specifications or abstract data type specifications. In a typical example the dependence on $\mathcal{S}_j$ comes from the signature of $\mathcal{S}_i$, which may involve a type

```
Begin list[ T : Type ] : Adt
  Constructor
    null : Carrier;
    cons( car, cdr ) : [T, Carrier] -> Carrier
End list

Begin InfTreeFin[ T : Type ] : ClassSpec
  Method
    branch : Self -> List[[T, Self]];
End InfTreeFin
```

Figure 4.13.: Trees of finite width and (possibly) infinite depth in CCSL (from (Hensel, 1999))

constructor $\mathsf{C}_{\mathcal{S}_j}$ whose semantics is a distinguished model of $\mathcal{S}_j$. However, it can also be the case that only some assertion of $\mathcal{S}_i$ uses a constructor of $\mathcal{S}_j$. Note that the use of *iterated* refers to iteration of different induction and coinduction principles that come into play in the described situation. Iterated does *not* refer to a mutual recursion of the specifications $\mathcal{S}_i$ and $\mathcal{S}_j$. In fact in CCSL the dependency relation between specifications is always a strict order.

An example of an iterated data type (that is an iterated specification without assertions) is the behavioural data type of trees of (possibly) infinite depth and arbitrary but finite width. The example appears originally in (Hensel, 1999). Figure 4.13 shows it in CCSL syntax. The crucial point here is, that the class specification InfTreeFin involves the type constructor List. This type constructor and its semantics is defined by the first specification List.

The functors that capture signatures of iterated specifications are usually called *data functors*. Data functors have been studied in (Jay, 1996; Cockett and Spencer, 1992) and also in (Hensel and Jacobs, 1997; Hensel, 1999; Rößiger, 2000a; Rößiger, 2000b). The work of Cockett and Spencer led to the categorical programming language CHARITY (Cockett and Fukushima, 1992). In a sense CCSL can be viewed as the specification language for CHARITY. The work of Hensel and Jacobs describes definition and proof principles for iterated data types under the assumption that suitable initial algebras and final coalgebras do exist in the base category. Rößiger proves that initial algebras and final coalgebras exist in **Set** for all (covariant) data functors. These latter two results are plugged together in this section.

The iterated specifications of CCSL are more general than the iterated data types that have been considered by Hensel, Jacobs, and Rößiger. Up to my knowledge, there are a number of open questions related to the semantics of iterated specifications. Therefore, a complete treatment of iterated specifications is beyond the scope of the present thesis.

The approach taken here is very pragmatic: Until better solutions are available, CCSL uses predicate lifting and relation lifting for abstract data types and behavioural types as described in (Hensel and Jacobs, 1997; Hensel, 1999). Because of the greater generality of iterated specifications in CCSL, the various liftings are not always well defined (for instance in case a class contains binary methods). In case they are not defined certain restrictions are imposed on iterated specifications (via improper ground signatures).

The first subsection describes the technicalities. The material is taken from (Hensel, 1999) and adopted to the setting of the present Chapter. I can only give the definitions here, for the rationale behind them I refer the reader to (Hensel and Jacobs, 1997) or (Hensel, 1999). The second subsection characterises those iterated specifications which have a well-defined semantics. The third subsection gives guidelines on how to ensure consistency for iterated specifications.

4.7.1. Semantics of Iterated Specifications

The technical means to allow type checking and semantics for iterated specifications are ground signatures. A CCSL specification consists of a finite list of entities $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n$, standing one after each other in one file. Each of the $\mathcal{S}_i$ is either a ground signature extension, a class specification, or an abstract data type specification. For each of the $\mathcal{S}_i$ there is a ground signature Ω_i and a model $\mathcal{M}_i$ of it, which are both not explicit in the CCSL source. The first pair $\langle \Omega_0, \mathcal{M}_0 \rangle$ consists of the empty ground signature (i.e., $|\Omega_0| = \emptyset$ and $\Omega_{0\sigma} = \emptyset$) and the empty model. Each of the $\mathcal{S}_i$ can define type constructors and constants. These items are added[20] to Ω_i and $\mathcal{M}_i$ to yield Ω_{i+1} and $\mathcal{M}_{i+1}$. Then Ω_{i+1} is used to type check $\mathcal{S}_{i+1}$ and $\mathcal{M}_{i+1}$ is used for the semantics of $\mathcal{S}_{i+1}$. This way a specification has access to (or can use) all the specifications and all the ground signature extensions that appear before it.

In the following I consider an arbitrary $\mathcal{S}$ from the finite list of $\mathcal{S}_i$ with associated ground signature Ω and a model $\mathcal{M}$ of Ω. For the three possibilities ($\mathcal{S}$ is a ground signature, an abstract data type, or a class specification) I describe which type constructors and constants are defined by $\mathcal{S}$ and what semantics they have. For some of the items a semantics can only be defined if the model $\mathcal{M}$ is proper and/or additional conditions hold. For these items I take the following approach: I first describe their semantics. If this is well defined, then the corresponding item is defined by $\mathcal{S}$ and added to the ground signature. The item stays undefined (and is not added to the ground signature) otherwise. This way it can happen that the ground signature Ω' (or its model) that is build from Ω and $\mathcal{S}$ is not proper, despite the fact that Ω (or its model) is proper. This has the described consequences for subsequent specifications.

The CCSL compiler deviates slightly from what I describe in the following. It generally uses only one interpretation for any given type variable, see Subsection 4.2.4.

[20] I disregard name clashes here. In the CCSL compiler later defined items hide earlier ones with the same name (there is no overloading).

Ground Signature Extensions

Let $\mathcal{S}$ be a ground signature extension. To state the obvious, $\mathcal{S}$ defines all items which are declared in $\mathcal{S}$. The semantics is taken from the environment or from the CCSL source as described in Subsection 4.3.

Abstract Data Type Specifications

Let $\mathcal{S}$ be an abstract data type specification with $\Bbbk$ type parameters $\alpha_1, \ldots, \alpha_\Bbbk$ and n constructor declarations $c_1 : \sigma_1, \ldots, c_n : \sigma_n$. Recall from page 151 that the combined constructor type of $\mathcal{S}$ is defined as $\sigma_\mathcal{S} = \mathcal{T}_C(c_1) + \cdots + \mathcal{T}_C(c_n)$. Recall also that the special type Self, which functions in the σ_i as place holder for the data type being defined, occurs in all the σ_i only strictly covariantly. Therefore I drop the contravariant argument, when considering the semantics of $\sigma_\mathcal{S}$. For data type specifications the variance of the type parameters is defined as

$$\mathcal{V}_x(\mathcal{S}) \quad \overset{\text{def}}{=} \quad \mathcal{V}_x(\sigma_\mathcal{S})$$

The following list describes what items are defined by the specification $\mathcal{S}$. In the description I use $\overline{U}$ to denote an arbitrary interpretation of the type variables $\overline{\alpha} = \alpha_1, \ldots, \alpha_\Bbbk$. For the interpretation of types positive and negative occurrences of the type variables are interpreted with different sets. However, for terms every type variable is interpreted by only one set. So depending on the context $\overline{U}$ denotes either $2\Bbbk$ sets $U_1^-, U_1^+, \ldots, U_\Bbbk^-, U_\Bbbk^+$ or only $\Bbbk$ sets $U_1, \ldots, U_\Bbbk$. Further I set $F_{\overline{U}}^\mathcal{S}(X) \overset{\text{def}}{=} [\![\sigma_\mathcal{S}]\!]_{\overline{U}}(X)$.

Type Constructor $\mathsf{C}_\mathcal{S}$ Let $\delta_{\overline{U}} : F_{\overline{U}}^\mathcal{S}(X_{\overline{U}}) \longrightarrow X_{\overline{U}}$ denote the initial $F_{\overline{U}}^\mathcal{S}$ algebra. If $\delta_{\overline{U}}$ exists for all interpretations $\overline{U}$, then the specification defines the type constructor $\mathsf{C}_\mathcal{S}$ of arity $\Bbbk$:

$$\vdash \mathsf{C}_\mathcal{S} \;::\; [\, \mathcal{V}_{\alpha_1}(\mathcal{S}); \ldots; \mathcal{V}_{\alpha_\Bbbk}(\mathcal{S})]$$

Its semantics is the carrier of the initial algebra:

$$[\![\mathsf{C}_\mathcal{S}]\!](\overline{U}) \quad = \quad X_{\overline{U}}$$

The action on morphisms can be defined via reduce, see below. None of the following items is defined, if $\delta_{\overline{U}}$ does not exist for all $\overline{U}$.

Constructors For every constructor declaration $c_i : \sigma_i$ the specification $\mathcal{S}$ defines a constant

$$\overline{\alpha} \;\mid\; \emptyset \;\vdash\; \mathsf{c}_i \;:\; \sigma_i[\,\mathsf{C}_\mathcal{S}[\overline{\alpha}] \;/\; \mathsf{Self}\,]$$

where $[\,\mathsf{C}_\mathcal{S}[\overline{\alpha}] \;/\; \mathsf{Self}\,]$ denotes the substitution of $\mathsf{C}_\mathcal{S}[\overline{\alpha}]$ for Self in σ_i. The semantics is

$$[\![\mathsf{c}_i]\!]_{\overline{U}} \quad = \quad \delta_{\overline{U}} \circ \kappa_i$$

where κ_i is the interpretation injection belonging to c_i (compare page 153).

Reduce The (higher-order) function reduce creates the unique algebra morphism out of the initial $[\![\sigma_{\mathcal{S}}]\!]$ algebra.

$$\overline{\alpha}, \beta \;\mid\; \emptyset \;\vdash\; \mathsf{reduce}_{\mathcal{S}} \;:\; (\sigma_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \sigma_n[\,\beta \,/\, \mathsf{Self}\,]) \;\Rightarrow\; \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \;\Rightarrow\; \beta$$

Let V be the interpretation of β and fix a list of functions $\overline{f} = f_1 : [\![\sigma_1]\!]_{\overline{U}}(V), \dots, f_n : [\![\sigma_n]\!]_{\overline{U}}(V)$. Note that the copairing $[f_1, \dots, f_n]$ is then a function with codomain V. Now $[\![\mathsf{reduce}]\!]_{\overline{U},V}(\overline{f})$ is defined as the unique function that makes the following diagram commute.

$$\begin{tikzcd}
F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}}) \arrow[rr, "\delta_{\overline{U}}"] \arrow[d, dashed, "F^{\mathcal{S}}_{\overline{U}}\left([\![\mathsf{reduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f})\right)"'] & & X_{\overline{U}} \arrow[d, dashed, "{[\![\mathsf{reduce}_{\mathcal{S}}]\!]_{\overline{U},V}(\overline{f})}"] \\
F^{\mathcal{S}}_{\overline{U}}(V) \arrow[rr, "{[f_1, \dots, f_n]}"] & & V
\end{tikzcd}$$

Map The map combinator gives the action of the functor $[\![\mathsf{C}_{\mathcal{S}}]\!]$ on morphisms. Recall from Section 4.3 that for a proper model $\mathcal{M}$ of the ground signature Ω the interpretation of the σ_i can be considered as a functor taking $2\Bbbk + 1$ arguments[21] Its action on morphisms is denoted with $[\![\sigma_i]\!]_{\overline{g}}(f)$ for suitable functions $\overline{g}$ and f. For each constructor c_i set $\widehat{\sigma}_i \overset{\text{def}}{=} \mathcal{T}_C(c_i)$. Let $\overline{V} = V_1^-, V_1^+, \dots, V_{\Bbbk}^-, V_{\Bbbk}^+$ be another interpretation of the type parameters $\alpha_1, \dots, \alpha_{\Bbbk}$ and let $\overline{g} = g_1^-, g_1^+, \dots, g_{\Bbbk}^-, g_{\Bbbk}^+$ be a list of functions such that

$$\begin{array}{lcl} g_1^- \;: V_1^- \longrightarrow U_1^- & & g_{\Bbbk}^- \;: V_{\Bbbk}^- \longrightarrow U_{\Bbbk}^- \\ & \cdots & \\ g_1^+ \;: U_1^+ \longrightarrow V_1^+ & & g_{\Bbbk}^+ \;: U_{\Bbbk}^+ \longrightarrow V_{\Bbbk}^+ \end{array}$$

Then, for proper models $\mathcal{M}$ of the ground signature, the semantics of $\mathsf{C}_{\mathcal{S}}$ is extended to a functor via

$$[\![\mathsf{C}_{\mathcal{S}}]\!](\overline{g}) \quad = \quad [\![\mathsf{reduce}_{\mathcal{S}}]\!]_{\overline{U},X_{\overline{V}}} \left([\![c_1]\!]_{\overline{V}} \circ [\![\widehat{\sigma}_1]\!]_{\overline{g}}(\mathrm{id}_{X_{\overline{V}}}),\; \dots,\; [\![c_n]\!]_{\overline{V}} \circ [\![\widehat{\sigma}_n]\!]_{\overline{g}}(\mathrm{id}_{X_{\overline{V}}})\right)$$

Case Distinction For each constructor c_i let $\widehat{\sigma}_i \overset{\text{def}}{=} \mathcal{T}_C(c_i)[\,\mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \,/\, \mathsf{Self}\,]$. The specification $\mathcal{S}$ defines case distinction as

$$\overline{\alpha}, \beta \;\mid\; \emptyset \;\vdash\; \mathsf{case}_{\mathcal{S}} \;:\; \left((\widehat{\sigma}_1 \Rightarrow \beta) \times \cdots \times (\widehat{\sigma}_n \Rightarrow \beta)\right) \;\Rightarrow\; \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \;\Rightarrow\; \beta$$

Let V be an interpretation for β, $f_1, \dots, f_n$ be a suitable vector of functions, and $x \in X_{\overline{U}}$. Then

$$[\![\mathsf{case}_{\mathcal{S}}]\!]_{\overline{U},V}(f_1, \dots, f_n)(x) \quad = \quad \begin{cases} \quad \vdots & \\ f_i(y) & \text{if } \exists y \in [\![\widehat{\sigma}_i]\!]_{\overline{U}} \;.\; x = \delta_{\overline{U}}(\kappa_i\, y) \\ \quad \vdots & \end{cases}$$

[21] The contravariant argument for Self is ignored here.

Recognisers For each constructor declaration $c_i : \sigma_i$ the specification $\mathcal{S}$ defines the recogniser

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{c?}_i : \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \Rightarrow \mathsf{Prop}$$

The semantics is

$$[\![\mathsf{c?}_i]\!]_{\overline{U}}(x) \quad = \quad [\![\mathsf{case}_{\mathcal{S}}]\!]_{\overline{U},\mathsf{bool}}(f_1, \dots, f_n)(x)$$

where $f_i = \lambda x \,.\, \top$ and $f_j = \lambda x \,.\, \bot$ for $j \neq i$.

Predicate Lifting $\mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}}$ For predicate lifting consider the following operator for a fixed list of $2\Bbbk$ parameter predicates $\overline{P}$.[22]

$$Q \subseteq X_{\overline{U}} \longmapsto \textstyle\coprod_{\delta_{\overline{U}}} \mathrm{Pred}([\![\sigma_{\mathcal{S}}]\!])(\overline{P}, Q, Q) \tag{4.4}$$

If this operator has a least fixed point, then the specification $\mathcal{S}$ defines predicate lifting as

$$\begin{aligned} \overline{\alpha} \mid \emptyset \vdash \mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}} \ : \ & (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow \\ & \quad (\alpha_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_{\mathcal{S}}[\alpha_1, \dots, \alpha_{\Bbbk}] \Rightarrow \mathsf{Prop} \end{aligned}$$

Its semantics is the least fixed point of (4.4).

Relation Lifting $\mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}}$ For relation lifting fix $2\Bbbk$ parameter relations (such that $R_i^+ \subseteq U_i^+ \times V_i^+$ and $R_i^- \subseteq U_i^- \times V_i^-$) and consider the following operator.

$$S \subseteq X_{\overline{U}} \times X_{\overline{V}} \longmapsto \textstyle\coprod_{\delta_{\overline{U}} \times \delta_{\overline{V}}} \mathrm{Rel}([\![\sigma_{\mathcal{S}}]\!])(\overline{R}, S, S) \tag{4.5}$$

If this has a least fixed point, then the specification $\mathcal{S}$ defines relation lifting as

$$\begin{aligned} \overline{\alpha}, \overline{\beta} \mid \emptyset \vdash \mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}} \ : \ & (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow (\alpha_{\Bbbk} \times \beta_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_{\Bbbk} \times \beta_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_{\mathcal{S}}[\alpha_1, \dots, \alpha_{\Bbbk}] \times \mathsf{C}_{\mathcal{S}}[\beta_1, \dots, \beta_{\Bbbk}] \Rightarrow \mathsf{Prop} \end{aligned}$$

The semantics of $\mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}}$ is the least fixed point of (4.5).

[22] (Hensel, 1999) defines this operator as $Q \subseteq X_{\overline{U}} \longmapsto (\delta_{\overline{U}}{}^{-1})^* \,\mathrm{Pred}(\cdots)$. However, the equation $(f^{-1})^* = \coprod_f$ holds in **Pred** for isomorphisms f.

Remark 4.7.1

1. In addition to the items above, the CCSL compiler also defines accessor functions. Accessor functions cannot be correctly typed in the type theory of the present chapter (see the remarks on this issue in Example 4.3.2 on page 145). For a constructor $c : \sigma^1 \times \cdots \times \sigma^m \Rightarrow \mathsf{Self}$ that takes m arguments there are m (partial) accessor functions
$$\overline{\alpha} \mid \emptyset \vdash \mathsf{acc}_c^j : \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \Rightarrow \sigma^j$$
They get semantics either as a partial function or as a dependently typed function via
$$[\![\mathsf{acc}_c^j]\!]_{\overline{U}}(x) \quad = \quad \begin{cases} y_j & \text{if } \exists y_1, \dots, y_m \,.\, \delta_{\overline{U}}(\kappa_c(y_1, \dots, y_m)) = x \\ \text{undefined} & \text{otherwise} \end{cases}$$
where κ_c is the interpretation injection for the constructor c.

2. The semantics of case can be defined via reduce. Consider the following diagram:
$$\begin{tikzcd}
F_{\overline{U}}^{\mathcal{S}}(X_{\overline{U}}) \arrow[r, dashed, "F_{\overline{U}}^{\mathcal{S}}(\mathsf{precase})"] \arrow[dd, "\delta_{\overline{U}}"'] & F_{\overline{U}}^{\mathcal{S}}(X_{\overline{U}} \times V) \arrow[d, "F_{\overline{U}}^{\mathcal{S}}(\pi_1)"] & \\
 & F_{\overline{U}}^{\mathcal{S}}(X_{\overline{U}}) \arrow[d, "\langle \delta_{\overline{U}}, {[f_1, \dots, f_n]} \rangle"] & \\
X_{\overline{U}} \arrow[r, dashed, "\mathsf{precase}"] & X_{\overline{U}} \times V \arrow[r, "\pi_2"] & V
\end{tikzcd}$$
By exploiting the arguments $f_1, \dots, f_n$ one can define an $F_{\overline{U}}^{\mathcal{S}}$ algebra on $X_{\overline{U}} \times V$ (on the right hand side in the preceding diagram). Initiality of $\delta_{\overline{U}}$ defines the unique function $\mathsf{precase}$, which makes the preceding diagram commute. Then one can define $[\![\mathsf{case}_{\mathcal{S}}]\!]_{\overline{U},V}(f_1, \dots, f_n) = \pi_2 \circ \mathsf{precase}$.

3. The predicate lifting for the abstract data type $\mathcal{S}$ is defined if the (full) predicate lifting for $[\![\sigma_{\mathcal{S}}]\!]$ is defined. This is for instance the case, if the ground signature Ω and its model $\mathcal{M}$ are proper. For the relation lifting an analogous statement holds. Both, the predicate lifting and the relation lifting, are computable functions.[23] The definition that I gave on the preceding pages does not describe an algorithm because I use a fixed point construction. The theorem provers PVS and

[23] A function is computable if there exists a algorithm (in the form of a Turing machine for instance) that can compute the function.

ISABELLE/HOL admit such definitions. However, in proofs it is often easier to work with definitions that describe a terminating algorithm (because then one can turn the definition into a terminating rewrite system). For this reason the CCSL compiler uses the following —equivalent— definitions for predicate and relation lifting of abstract data types.

For predicate lifting the CCSL compiler outputs the following (I take the liberty to omit the technical noise of the $\overline{U}$ and of $[\![-]\!]$):

$$\mathsf{Pred}_{\mathsf{C}_\mathcal{S}}(\overline{P}) \quad = \quad \mathsf{reduce}_\mathcal{S}\big(\ \mathrm{Pred}([\![\sigma_1]\!])\ (\overline{P}, \mathsf{tt}, \mathsf{tt}), \ldots, \mathrm{Pred}([\![\sigma_n]\!])\ (\overline{P}, \mathsf{tt}, \mathsf{tt})\ \big)$$

Here $\mathsf{tt} \subseteq \mathsf{bool} \overset{\mathrm{def}}{=} \{\top\}$ is the predicate that holds only for true.

The relation lifting for abstract data types is slightly more complicated to define. The idea is as follows: From an element $u \in X_{\overline{U}}$ one computes a function $f \in X_{\overline{V}} \Rightarrow \mathsf{bool}$ such that for $v \in X_{\overline{V}}$ one has $f\,v = \top$ if and only if $[\![\mathsf{Rel}_{\mathsf{C}_\mathcal{S}}]\!]_{\overline{U},\overline{V}}(\overline{R})(u, v)$. As definition mechanism for functions with domain $X_{\overline{U}}$ there is only reduce available, therefore one needs an algebra acting on $X_{\overline{V}} \Rightarrow \mathsf{bool}$. This algebra is defined with a particular instantiation of the operator for relation lifting. Recall from Definition 4.4.7 (2) that relation lifting is a function of the following type

$$\mathrm{Rel}(\tau)(\overline{R}, S) \quad : \quad [\![\tau]\!]_{\overline{U}}(Y) \ \times\ [\![\tau]\!]_{\overline{V}}(X) \ \longrightarrow\ \mathsf{bool} \qquad (*)$$

where $\overline{R}$ is a list of parameter relations $R_i \subseteq U_i \times V_i$ and $S \subseteq Y \times X$ (I ignore the contravariant argument relation for Self). Define now

$$\begin{aligned} \mathrm{Rel}'(\tau)(\overline{R}) \quad &: \quad [\![\tau]\!]_{\overline{U}}(X \Rightarrow \mathsf{bool}) \ \times\ [\![\tau]\!]_{\overline{V}}(X) \ \longrightarrow\ \mathsf{bool} \\ \mathrm{Rel}'(\tau)(\overline{R}) \quad &\overset{\mathrm{def}}{=} \quad \mathrm{Rel}(\tau)(\overline{R},\ \ \lambda f \in X \Rightarrow \mathsf{bool},\ y \in X\ .\ f\,y) \end{aligned}$$

That is, in $(*)$ one takes $Y = X \Rightarrow \mathsf{bool}$ and uses an S that performs function application. Now, for any constructor $c_i : \sigma_i \Rightarrow \mathsf{Self}$, there is the following function

$$\begin{aligned} \mathrm{Rel}_{c_i}(\overline{R}) \quad &: \quad [\![\sigma_i]\!]_{\overline{U}}(X_{\overline{V}} \Rightarrow \mathsf{bool}) \ \longrightarrow\ X_{\overline{V}} \Rightarrow \mathsf{bool} \\ \mathrm{Rel}_{c_i}(\overline{R}) \quad &\overset{\mathrm{def}}{=} \quad \lambda f : [\![\sigma_i]\!]_{\overline{U}}(X_{\overline{V}} \Rightarrow \mathsf{bool})\ .\ \lambda y : X_{\overline{V}}\ . \\ &\qquad\qquad [\![\mathsf{c?}_i]\!]_{\overline{V}}(y) \ \ \wedge\ \ \mathrm{Rel}'([\![\sigma_i]\!])(\overline{R})(f, {\delta_{\overline{V}}}^{-1}\,y) \end{aligned}$$

Here $\wedge$ should be evaluated in a non-strict way: if the recogniser $\mathsf{c?}_i$ returns false then the result is false. Otherwise the inverse of the algebra $\delta_{\overline{V}}$ delivers something in $[\![\sigma_i]\!]_{\overline{V}}(X_{\overline{V}})$, as required by Rel'. Note that the Rel_{c_i} form an $\mathcal{S}$ algebra on $X_{\overline{V}} \Rightarrow \mathsf{bool}$ and can therefore be passed as argument to reduce:

$$[\![\mathsf{Rel}_{\mathsf{C}_\mathcal{S}}]\!]_{\overline{U},\overline{V}}(\overline{R})(u, v) \quad = \quad [\![\mathsf{reduce}_\mathcal{S}]\!]_{\overline{U}, X_V \Rightarrow \mathsf{bool}}\big(\ \cdots \mathrm{Rel}_{c_i}(\overline{R}) \cdots\ \big)\ u\ v$$

This definition yields the least fixed point of 4.5.

```
Begin list[ A : Type ] : ADT
   Constructor
      null : Carrier;
      cons( car, cdr ) : [A, Carrier] -> Carrier
End list
```

Figure 4.14.: The data type of lists from the CCSL prelude

```
RelEvery(R: [[U , V] -> bool])    :   [[list[U] , list[V]] -> bool] =
  Lambda (u: list[U] , v: list[V]) :
    reduce[U, [list[V] -> bool]]
      (null?[V] ,
       Lambda (x: U , y: [list[V] -> bool]) :   Lambda (l: list[V]) :
          cons?[V](l) And R(x , car[V](l)) And y(cdr[V](l)))
      (u)(v)
```

Figure 4.15.: The relation lifting for lists, generated by the CCSL compiler

As an example for this mind twisting definition I show in Figure 4.15 what the CCSL compiler generates as relation lifting for the abstract data type of lists from Figure 4.14. The relation lifting is called RelEvery there and instead of the inverted list algebra the CCSL compiler uses the two accessors car and cdr. All instantiations are given in square brackets after the identifier.

Example 4.7.2 This example shows the items that are defined by the abstract data type specification of lists from the CCSL prelude. For convenience Figure 4.14 repeats the CCSL source code. It defines the type constructor

$$\vdash \mathsf{list} \;:\; [(?,0)]$$

Its semantics is the functor $\mathsf{list} : \mathbf{Set} \longrightarrow \mathbf{Set}$ that maps every set A to the initial list algebra $[\mathsf{nil}_A, \mathsf{cons}_A] : \mathbf{1} + A \times A^* \longrightarrow A^*$, where A^* is the set of finite words over A. As before I ignore the argument for the negative occurrences of A.

For a function $f : X \longrightarrow Y$ the action of the functor $\mathsf{list}(f) : X^* \longrightarrow Y^*$ (i.e., the map combinator for lists) is defined as

$$\begin{aligned} \mathsf{list}(f)(\mathsf{nil}_X) &= \mathsf{nil}_Y \\ \mathsf{list}(f)(\mathsf{cons}_X(x, l)) &= \mathsf{cons}(f\,x, \mathsf{list}(f)(l)) \end{aligned}$$

In the context of lists, reduce is sometimes called foldright. For a constant $y \in Y$ and a function $g : X \times Y \longrightarrow Y$ it is defined as

$$\begin{array}{rcl} \mathsf{reduce}(y, g)(\mathsf{nil}_X) & = & y \\ \mathsf{reduce}(y, g)(\mathsf{cons}_X(x, l)) & = & g(x, \mathsf{reduce}(y, g)(l)) \end{array}$$

For a parameter predicate $P \subseteq X$ the predicate lifting $\mathsf{Pred}_{\mathsf{list}}(P) \subseteq X^*$ is

$$\begin{array}{rcl} \mathsf{Pred}_{\mathsf{list}}(P)(\mathsf{nil}_X) & = & \top \\ \mathsf{Pred}_{\mathsf{list}}(P)(\mathsf{cons}_X(x, l)) & = & P\,x \;\wedge\; \mathsf{Pred}_{\mathsf{list}}(P)(l) \end{array}$$

For relation lifting one has to stare for a while at Figure 4.15 to see that it is equivalent with the following characterisation (for $R \subseteq U \times V$)

$$\mathsf{Rel}_{\mathsf{list}}(R)(l_1, l_2) \quad = \quad \begin{cases} \top & \text{if } l_1 = \mathsf{nil}_U \;\wedge\; l_2 = \mathsf{nil}_V \\ R(u, v) \;\wedge\; \mathsf{Rel}_{\mathsf{list}}(R)(l_1', l_2') & \begin{array}{l}\text{if } l_1 = \mathsf{cons}_U(u, l_1') \\ \qquad \wedge\; l_2 = \mathsf{cons}_V(v, l_2')\end{array} \\ \bot & \text{otherwise} \end{cases} \qquad \blacksquare$$

Coalgebraic Class Specifications

I turn now to the description of the items that class specifications contribute to the current ground signature.

Let $\mathcal{S}$ be a class specification over the ground signature Ω with a model $\mathcal{M}$ of Ω. Assume that the signature of $\mathcal{S}$ contains $\Bbbk$ type parameters $\overline{\alpha} = \alpha_1, \dots, \alpha_{\Bbbk}$, n method declarations $m_i : \tau_i$, and m constructor declarations $c_j : \sigma_j$. Subsection 4.4 defines (on page 151) the combined method type of $\mathcal{S}$ as $\tau_{\mathcal{S}} = \mathcal{T}_M(\sigma_1) \times \cdots \times \mathcal{T}_M(\sigma_n)$ and the combined constructor type $\sigma_{\mathcal{S}} = \mathcal{T}_C(\sigma_1) + \cdots + \mathcal{T}_C(\sigma_m)$. The variance of the type parameters and of Self in $\mathcal{S}$ is defined as

$$\mathcal{V}_x(\mathcal{S}) \quad \overset{\text{def}}{=} \quad \mathcal{V}_x(\tau_{\mathcal{S}})$$

What items are defined by $\mathcal{S}$ in the following depends (among other things) on whether $\mathcal{S}$ is processed with final or loose semantics. Loose semantics is the default, final semantics can be chosen with the keyword `FINAL`, see Subsection 4.4.2. Like on the preceding pages I use $\overline{U}$ to denote an arbitrary interpretation of the type variables $\overline{\alpha}$ and also $F^{\mathcal{S}}_{\overline{U}}(Y, X) = [\![\tau_{\mathcal{S}}]\!]_{\overline{U}}(Y, X)$.

Type Constructor $\mathsf{C}_{\mathcal{S}}$ The specification $\mathcal{S}$ defines the type constructor $\mathsf{C}_{\mathcal{S}}$ of arity $\Bbbk$:

$$\vdash \mathsf{C}_{\mathcal{S}} \;::\; [\,\mathcal{V}_{\alpha_1}(\mathcal{S}); \dots; \mathcal{V}_{\alpha_{\Bbbk}}(\mathcal{S})]$$

The semantics of $\mathsf{C}_{\mathcal{S}}$ should be a functor taking $2\Bbbk$ arguments (compare Definition 4.2.5 on page 139). However, in many cases this functor is not fully defined.

The morphism component of $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is only defined under the following two conditions: The specification $\mathcal{S}$ must request final semantics and $\mathcal{S}$ must not contain any assertions. If the specification $\mathcal{S}$ does contain assertions then the object component of $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is only defined if the respective arguments for positive and negative occurrences are equal, that is if $U_i^- = U_i^+$, regardless whether final or loose semantics is used. In the following I simply state the definitions without repeating these side conditions again.

For final semantics let $\varepsilon_{\overline{U}} : X_{\overline{U}} \longrightarrow F^{\mathcal{S}}_{\overline{U}}(X_{\overline{U}}, X_{\overline{U}})$ denote the final $F^{\mathcal{S}}_{\overline{U}}$ coalgebra satisfying the assertions of $\mathcal{S}$.[24] For every possible $\overline{U}$ choose $\delta_{\overline{U}}$ such that $(\langle X_{\overline{U}}, \varepsilon_{\overline{U}}, \delta_{\overline{U}}\rangle)_{\overline{U}}$ is a model of $\mathcal{S}$.

For loose semantics choose an arbitrary model $(\langle X_{\overline{U}}, \varepsilon_{\overline{U}}, \delta_{\overline{U}}\rangle)_{\overline{U}}$ of $\mathcal{S}$.

Then

$$[\![\mathsf{C}_{\mathcal{S}}]\!](\overline{U}) \quad = \quad X_{\overline{U}}$$

If the specification $\mathcal{S}$ contains no assertions and if non of the α_i has mixed variance, then for final semantics the mapping $[\![\mathsf{C}_{\mathcal{S}}]\!]$ is extended to a functor, see Item Map below.

Methods For each method declaration $m_i : \tau_i$ the specification $\mathcal{S}$ defines a symbol

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{m}_i : \tau_i[\,\mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \,/\, \mathsf{Self}\,]$$

Note that τ_i is a method type, so it can be decomposed into $\tau_i = (\mathsf{Self} \times \tau_i') \Rightarrow \tau_i''$. The semantics of m_i is (as defined in Definition 4.5.3):

$$[\![\mathsf{m}_i]\!]_{\overline{U}} \quad = \quad \lambda x : X_{\overline{U}}, p : [\![\tau_i']\!]_{\overline{U}}(X_{\overline{U}}, X_{\overline{U}}) \;.\; \pi_i(\varepsilon_{\overline{U}}(x))\,(p)$$

Here π_i is the interpretation projection belonging to the method m_i (see page 153).

Constructors For each constructor declaration $c_j : \sigma_j$ the specification $\mathcal{S}$ defines a symbol

$$\overline{\alpha} \mid \emptyset \vdash \mathsf{c}_j : \sigma_j[\,\mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \,/\, \mathsf{Self}\,]$$

By definition σ_j is a constant constructor type, so $\sigma_j = \sigma_j' \Rightarrow \mathsf{Self}$. Now

$$[\![\mathsf{c}_j]\!]_{\overline{U}} \quad = \quad \delta_{\overline{U}} \circ \kappa_j$$

Coreduce If $\mathcal{S}$ is processed with final semantics then there is a (higher-order) function $\mathsf{coreduce}$ (sometimes also called unfold) that creates the unique morphism into the final coalgebra.

$$\overline{\alpha}, \beta \mid \emptyset \vdash \mathsf{coreduce}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \beta \Rightarrow \mathsf{C}_{\mathcal{S}}[\overline{\alpha}]$$

[24] Note that such $\varepsilon_{\overline{U}}$ might exist, even in case where there is no final coalgebra for $F^{\mathcal{S}}_{\overline{U}}$.

For the semantics fix an interpretation V of β and let $\overline{f} = f_1, \dots, f_n$ be a list of functions such that $f_i : \llbracket \tau_i \rrbracket_{\overline{U}}(V, V)$. With some shuffling one can transform the f_i into a $F^{\mathcal{S}}_{\overline{U}}$ coalgebra on state space V, denoted with $\langle \overline{f} \rangle$. The semantic of $\mathsf{coreduce}_{\mathcal{S}}$ is only defined for those f_i for which $\langle \overline{f} \rangle$ fulfils the method assertions of $\mathcal{S}$. If this is the case then $\llbracket \mathsf{coreduce} \rrbracket_{\overline{U},V}(f_1, \dots, f_n)$ is the unique function that lets the following diagram commute.

$$\begin{tikzcd}
V \arrow[rr, dashed, "\llbracket \mathsf{coreduce}_{\mathcal{S}} \rrbracket_{\overline{U},V}(\overline{f})"] \arrow[d, "\langle \overline{f} \rangle"'] & & X_{\overline{U}} \arrow[d, "\varepsilon_{\overline{U}}"] \\
F^{\mathcal{S}}_{\overline{U}}(V, V) \arrow[dr, dashed, "F^{\mathcal{S}}_{\overline{U}}(V, \mathsf{coreduce}_{\mathcal{S}} \rrbracket_{\overline{U},V}(\overline{f}))"'] & & F^{\mathcal{S}}_{\overline{U}}(X, X) \arrow[dl, dashed, "F^{\mathcal{S}}_{\overline{U}}(\llbracket \mathsf{coreduce}_{\mathcal{S}} \rrbracket_{\overline{U},V}(\overline{f}), X)"] \\
& F^{\mathcal{S}}_{\overline{U}}(V, X) &
\end{tikzcd}$$

Map For final semantics the morphism part of $\llbracket \mathsf{C}_{\mathcal{S}} \rrbracket$ is defined under the following conditions: First, the model $\mathcal{M}$ of the ground signature must be proper. Second $\mathcal{S}$ must not contain any assertions.

If these conditions are met, the interpretation of the method types τ_i can be regarded as a functor taking $2\Bbbk + 2$ arguments, whose morphism part is denoted with $\llbracket \tau_i \rrbracket_{\overline{g}}(f^-, f^+)$. The morphism part of $\llbracket \mathsf{C}_{\mathcal{S}} \rrbracket$ can now be defined via $\mathsf{coreduce}$: Fix a second interpretations for the type parameters $\overline{V} = V_1^-, V_1^+, \dots, V_{\Bbbk}^-, V_{\Bbbk}^+$ and assume a vector of functions

$$\begin{array}{ll} g_1^- : U_1^- \longrightarrow V_1^- & \\ g_1^+ : V_1^+ \longrightarrow U_1^+ & \end{array} \quad \cdots \quad \begin{array}{ll} g_{\Bbbk}^- : U_{\Bbbk}^- \longrightarrow V_{\Bbbk}^- \\ g_{\Bbbk}^+ : U_{\Bbbk}^+ \longrightarrow V_{\Bbbk}^+ \end{array}$$

and set $\overline{g} = g_1^-, g_1^+, \dots, g_{\Bbbk}^-, g_{\Bbbk}^+$. Then

$$\llbracket \mathsf{C}_{\mathcal{S}} \rrbracket(\overline{g}) \quad = \quad \llbracket \mathsf{coreduce}_{\mathcal{S}} \rrbracket_{\overline{U},X_{\overline{V}}}\big(\cdots \; \llbracket \tau_i \rrbracket_{\overline{g}}(\mathrm{id}_{X_{\overline{V}}}, \mathrm{id}_{X_{\overline{V}}}) \; (\llbracket \mathsf{m}_i \rrbracket_{\overline{V}}) \; \cdots \big)$$

Invariant Recogniser The invariant recogniser for $\mathcal{S}$ is a functional that takes a signature model of $\mathcal{S}$ and a predicate on the state space of that signature model as arguments. It returns true if the predicate is an invariant (for the signature model) according to Definition 4.4.10 (1). The invariant recogniser is defined whenever the ground signature Ω is proper. Its type is as follows.

$$\overline{\alpha}, \beta \;\mid\; \emptyset \;\vdash\; \mathsf{invariant}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \\ (\beta \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{Prop}$$

If ⟨`class`⟩ is the name of $\mathcal{S}$ in the CCSL source code then the CCSL compiler generates the identifier ⟨`class`⟩`_class_invariant?` for the invariant recogniser.

Bisimulation Recogniser The bisimulation recogniser takes two signature models and a relation as arguments. It returns true if the relation is a bisimulation according to Definition 4.4.10 (3). The type of the bisimulation recogniser is

$$\begin{aligned}\overline{\alpha}, \beta, \gamma \;\mid\; \emptyset \;\vdash\; \mathsf{bisimulation}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \\ (\tau_1[\,\gamma \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\gamma \,/\, \mathsf{Self}\,]) \Rightarrow (\beta \times \gamma \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{Prop}\end{aligned}$$

The compiler uses ⟨`class`⟩`_class_bisimulation?` as identifier for $\mathsf{bisimulation}_{\mathcal{S}}$.

Morphism Recogniser The morphism recogniser returns true for functions that are coalgebra morphisms in the sense of Definition 3.2.2 (on page 80). Its type is

$$\begin{aligned}\overline{\alpha}, \beta, \gamma \;\mid\; \emptyset \;\vdash\; \mathsf{morphism}_{\mathcal{S}} : (\tau_1[\,\beta \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\beta \,/\, \mathsf{Self}\,]) \Rightarrow \\ (\tau_1[\,\gamma \,/\, \mathsf{Self}\,] \times \cdots \times \tau_n[\,\gamma \,/\, \mathsf{Self}\,]) \Rightarrow (\beta \Rightarrow \gamma) \Rightarrow \mathsf{Prop}\end{aligned}$$

The CCSL compiler generates the name ⟨`class`⟩`_class_morphism?` for it.

The CCSL compiler treats the three recognisers for invariants, bisimulations, and morphisms special. They are added to the ground signature after processing the signature of $\mathcal{S}$ such that one can use these recognisers in method and constructor assertions.

Predicate Lifting $\mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}}$ For predicate lifting consider the following operator

$$Q \subseteq X_{\overline{U}} \longmapsto \varepsilon^{*}_{\overline{U}} \; \mathrm{Pred}(\llbracket \tau_{\mathcal{S}} \rrbracket)(\overline{P}, \top_{X_{\overline{U}}}, Q) \tag{4.6}$$

where $\overline{P} = P^-_1, P^+_1, \ldots, P^-_{\Bbbk}, P^+_{\Bbbk}$ are $2\Bbbk$ parameter predicates. If (4.6) has a greatest fixed point for all parameter predicates then the specification $\mathcal{S}$ defines the constant for predicate lifting as

$$\begin{aligned}\overline{\alpha} \;\mid\; \emptyset \;\vdash\; \mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}} \;:\; & (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ & (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow \\ & (\alpha_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \Rightarrow \mathsf{Prop}\end{aligned}$$

The semantics of $\mathsf{Pred}_{\mathsf{C}_{\mathcal{S}}}$ is the greatest fixed point of (4.6).

Relation Lifting $\mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}}$ For relation lifting consider the following operator for a suitable list of parameter relations $\overline{R}$ (with $R^+_i \subseteq U^+_i \times V^+_i$ and $R^-_i \subseteq U^-_i \times V^-_i$):

$$S \subseteq X_{\overline{U}} \times X_{\overline{V}} \longmapsto (\varepsilon^{*}_{\overline{U}} \times \varepsilon^{*}_{\overline{V}}) \; \mathrm{Rel}(\llbracket \tau_{\mathcal{S}} \rrbracket)(\overline{R}, S, S) \tag{4.7}$$

```
Begin Sequence[ A : Type ] : ClassSpec
  Method
    next : Self -> Lift[[A,Self]];
End Sequence
```

Figure 4.16.: Possibly infinite queues in CCSL

If it has a greatest fixed point then there is a constant for relation lifting of the following type

$$\begin{aligned}\overline{\alpha}, \overline{\beta} \mid \emptyset \vdash \mathsf{Rel}_{\mathsf{C}_{\mathcal{S}}} &: (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_1 \times \beta_1 \Rightarrow \mathsf{Prop}) \Rightarrow \\ &(\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow (\alpha_2 \times \beta_2 \Rightarrow \mathsf{Prop}) \Rightarrow \cdots \Rightarrow (\alpha_{\Bbbk} \times \beta_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \\ &\qquad (\alpha_{\Bbbk} \times \beta_{\Bbbk} \Rightarrow \mathsf{Prop}) \Rightarrow \mathsf{C}_{\mathcal{S}}[\overline{\alpha}] \times \mathsf{C}_{\mathcal{S}}[\overline{\beta}] \Rightarrow \mathsf{Prop}\end{aligned}$$

Its semantics is the greatest fixed point of (4.7).

In general, predicate and relation lifting for class specifications is only semi decidable.[25] So it is impossible to give an algorithmic description for the liftings of class specification. The CCSL compiler outputs definitions that rely on a the Knaster/Tarski characterisation of fixed points in complete lattices (Tarski, 1955).

Example 4.7.3 The queue example is not well-suited for illustration here because its type parameter occurs with mixed variance, so for the queue specification not all items are fully defined. Let me therefore reconsider the example of possibly infinite sequences from Section 2.6. Its rather short CCSL version is in Figure 4.16.

The sequence signature contains one type parameter occurring strictly covariant, therefore

$$\vdash \mathsf{Sequence} :: [(?, 0)]$$

The final model for sequences is described in Subsection 2.6.7 on page 69. The interpretation for the type constructor Sequence is defined for all interpretations U^-, U^+ of the type parameter A. However, since the type parameter A has positive variance, the argument U^- is ignored:

$$\begin{aligned}[\![\mathsf{Sequence}]\!](U^-, U^+) \quad &= \quad \mathsf{Seq}[U^+] \\ &= \quad \{f : \mathbb{N} \longrightarrow \mathsf{Lift}[U^+] \mid \forall n \in \mathbb{N} \,.\, f(n) = \mathsf{bot} \;\text{ implies }\; \forall m > n \,.\, f(m) = \mathsf{bot}\}\end{aligned}$$

[25] A predicate P is semi decidable if there exists an algorithm with the following properties. If the algorithm gets $x \in P$ as input then it terminates with result „yes“. On an input $x \notin P$ it terminates with „no“ or runs forever, see (U. Schöning, 1997). In particular the characteristic function of a semi decidable predicate cannot be computable.

where bot is the constant associated with the abstract data type Lift, see Example 4.3.2 (on page 144). In the following I silently ignore the argument for the negative occurrences of the type parameter A.

The interpretation of the method next is as described in Equation 2.8 (on page 69). The interpretation of $\mathsf{coreduce}$ is given by the unique function ! from Equation 2.9. Let g be a function $Y \longrightarrow U \times Y + \mathbf{1}$, then $h = [\![\mathsf{coreduce}]\!]_{U,Y}(g) : Y \longrightarrow \mathsf{Seq}[U]$ is the unique function for which the following equation holds:

$$[\![\mathsf{next}]\!]_U(h\,y) \quad = \quad \begin{cases} \mathsf{bot} & \text{if } g\,y = \mathsf{bot} \\ \mathsf{up}(u,\ h\,y') & \text{if } g\,y = \mathsf{up}(u, y') \end{cases}$$

(As an alternative to the definition of $\mathsf{coreduce}$ for a concrete representation of the final model one can use the preceding equation with a lazy evaluation scheme as the definition of $\mathsf{coreduce}$. This is in fact what the experimental programming language Charity (Cockett and Fukushima, 1992) does.)

Let me turn to the morphism part of $[\![\mathsf{Sequence}]\!]$ now. Assume a function $g : V \longrightarrow U$. Then $[\![\mathsf{Sequence}]\!](g)$ is a function $\mathsf{Seq}[V] \longrightarrow \mathsf{Seq}[U]$, which is defined as

$$[\![\mathsf{Sequence}]\!](g)\ f\ n \quad = \quad \begin{cases} \mathsf{bot} & \text{if } f\,n = \mathsf{bot} \\ \mathsf{up}(g\,v) & \text{if } f\,n = \mathsf{up}\,v \end{cases}$$

The predicate lifting $\mathsf{Pred}_{\mathsf{Sequence}}(P)$, for a parameter predicate $P \subseteq U$, is the greatest predicate Q with the following property

$$Q(f) \quad \text{if and only if} \quad \begin{cases} [\![\mathsf{next}]\!]_U(f) = \bot & \text{or} \\ [\![\mathsf{next}]\!]_U(f) = \mathsf{up}(u, f') \ \wedge\ P(u) \ \wedge\ Q(f') \end{cases}$$

for all $f \in \mathsf{Seq}[U]$. It is easy to see that

$$\mathsf{Pred}_{\mathsf{Sequence}}(P)(f) \qquad \text{if and only if} \qquad \forall n\,.\, f\,n = \mathsf{up}(u) \ \text{ implies } \ P\,u$$

The relation lifting $\mathsf{Rel}_{\mathsf{Sequence}}(R)$, for a parameter relation $R \subseteq U \times V$, is the greatest relation $S \subseteq \mathsf{Seq}[U] \times \mathsf{Seq}[V]$ such that

$$S(f,g) \quad \text{if and only if} \quad \begin{cases} [\![\mathsf{next}]\!]_U(f) = \bot \ \wedge\ [\![\mathsf{next}]\!]_V(g) = \bot & \text{or} \\ [\![\mathsf{next}]\!]_U(f) = \mathsf{up}(u, f') \ \wedge\ [\![\mathsf{next}]\!]_V(g) = \mathsf{up}(v, g') \\ \qquad\qquad \wedge\ R(u,v) \wedge S(f',g') \end{cases}$$

Again, for the concrete final model, it is easy to see that

$$\mathsf{Rel}_{\mathsf{Sequence}}(R)(f,g) \quad \text{if and only if} \quad \forall n\,.\begin{cases} f\,n = g\,n = \bot & \text{or} \\ f\,n = \mathsf{up}(u) \ \wedge\ g\,n = \mathsf{up}(v) \ \wedge\ R(u,v) \end{cases} \quad \blacksquare$$

4.7.2. Iterated Specifications for Polynomial Functors

The general case of iterated specifications is not completely understood yet. For instance, it is unclear how to get a functorial semantics of the queue specification from Example 4.5.15. Further, the case where iterated specifications contain class specifications with binary methods has not been investigated. Only the case of polynomial functors has been investigated in (Hensel and Jacobs, 1997; Hensel, 1999; Rößiger, 2000b). The result is the following theorem.

Theorem 4.7.4 *Let $\langle \mathcal{S}_i, \Omega_i, \mathcal{M}_i \rangle_{i \leq n}$ be a finite list of triples, where each $\mathcal{S}_i$ is either a ground signature extension, a coalgebraic class specification, or an abstract data type specification, and where the Ω_i and the $\mathcal{M}_i$ are constructed as described on the preceding pages. Assume that all the $\mathcal{S}_i$ comply with the following conditions:*

- *If $\mathcal{S}_i$ is a ground signature extension, then $\mathcal{S}_i$ and its model are proper. Further, all type constructors of $\mathcal{S}_i$ are type constants (i.e., have arity zero).*
- *If $\mathcal{S}_i$ is a coalgebraic class specification, then*
  - *all its type parameters have strictly positive variance,*
  - *all its method types are polynomial,*
  - *it specifies final semantics via the keyword* `FINAL`*,*
  - *if $\mathcal{S}_i$ contains assertions then all $\mathcal{S}_j$ with $j > i$ use the type constructor $\mathsf{C}_{\mathcal{S}_i}$ only with constant arguments,*
  - *all method assertions and all constructor assertions of $\mathcal{S}_i$ are invariant with respect to behavioural equality,*
  - *$\mathcal{S}_i$ is consistent.*
- *If $\mathcal{S}_i$ is an abstract data type specification, then all its type parameters have strictly positive variance.*

If these conditions hold then all Ω_i and all $\mathcal{M}_i$ are proper with one exception: The morphism component of some class specifications might be undefined. In particular, there exist initial models for all data type specifications and final models for all class specifications among the $\mathcal{S}_i$.

Further the following two technical conditions are fulfilled for all type constructors C: The interpretation of the relation lifting $\mathsf{Rel}_{\mathsf{C}}$ is fibred (over the interpretation of C) and it commutes with equality.

Before I can tackle the proof I have to generalise a few results. For the following two lemmas let $\mathcal{M}$ be a proper model of a proper ground signature Ω such that all relation liftings in $\mathcal{M}$ are fibred and commute with equality. By induction on the structure of types as in Lemma 2.6.9 (2) and (7) we get the next lemma.

Lemma 4.7.5 *Let τ be a polynomial type over Ω. Then the relation lifting of τ is fibred and commutes with equality.* □

Now also the proof of Proposition 2.6.12 can be generalised:

Lemma 4.7.6 *Let τ be a polynomial type over Ω. Then τ coalgebra morphisms are functional bisimulations.* □

Proof (of Theorem 4.7.4) The proof goes by induction on i and coalesces Proposition 4.5.18 of the present thesis with the results of Chapter 4 of (Hensel, 1999) and Chapter 6 of (Rößiger, 2000b). For $i = 0$ the proposition holds trivially, because Ω_0 is the empty ground signature and $\mathcal{M}_0$ the empty model. In the induction step there are three possibilities:

- If $\mathcal{S}_i$ is a ground signature extension, then the assumptions on ground signatures guarantees that Ω_i and $\mathcal{M}_i$ are proper. The relation lifting of the type constants in $\mathcal{S}_i$ fulfils the technical conditions trivially, because the relation lifting for constants takes no arguments.

- Let $\mathcal{S}_i$ be a coalgebraic class specification over signature Σ with combined method type τ. All type constructors in Ω_{i-1} are either constants, least fixed points (stemming from abstract data type specifications), or greatest fixed points (stemming from coalgebraic class specifications). Therefore the semantics of τ is a data functor in the sense of Rößiger and Hensel. Rößiger's Lemma 6.2.7 gives the final τ coalgebra as a set of labelled elementary trees.

  If $\mathcal{S}_i$ does not contains any assertions then its semantics is fully defined (including the morphism component).

  In case $\mathcal{S}_i$ does contain assertions then, by Lemma 4.7.6, τ coalgebra morphisms preserve the validity of the method assertions of $\mathcal{S}_i$ and we can construct the final model of $\mathcal{S}_i$ as in Proposition 4.5.18. This gives the semantics of any constant instantiation of $\mathcal{S}_i$ in subsequent specifications. The morphism component of the semantics of $\mathcal{S}_i$ is not used and stays undefined.

  The proof of this case is finished with Hensel's results: His Theorem 4.8 proves the existence of predicate and relation lifting, Proposition 4.9 shows that relation lifting is fibred, and Lemma 4.22 that it commutes with equality.

- Let $\mathcal{S}_i$ be an algebraic class specification with combined constructor type σ. Again the semantics of σ is a data functor and the initial σ algebra exists by Rößiger's Lemma 6.2.6. Then Hensel's Theorem 4.8 shows that predicate and relation lifting for $\mathcal{S}_i$ exist, Proposition 4.9 shows that relation lifting is fibred, and by Lemma 4.18 it commutes with equality. □

In this subsection I combined results from (Hensel and Jacobs, 1997) and (Rößiger, 2000b) to characterise the fragment of CCSL for which (at the time of writing) the semantics is well defined. The CCSL compiler provides the `-pedantic` switch (see Subsection 4.9.9 on page 223) for checking whether a specification lies within the well defined fragment of CCSL.

Note that even the simple queue specification from Figure 4.11 does not fulfil the assumptions of the preceding theorem because it has a type parameter with mixed variance. This shows that there is still a need for more general results on the existence of initial algebras and final coalgebras.

4.7.3. Using CCSL consistently

The preceding theorem 4.7.4 proves that the semantics of CCSL is well defined for polynomial functors and their iterations. As long as one stays in this fragment one can only introduce inconsistencies by writing an inconsistent specification. Interesting examples lead often beyond the assumptions of Theorem 4.7.4: Already the queue specification of this chapter contains a contravariant type variable and does therefore not fit into the preceding theorem.

To cope with the general situation, the CCSL compiler is very carefully constructed such that a few guide lines suffice to ensure consistency. For instance, when the CCSL compiler generates the relation lifting of a class specification it uses a greatest fixed point construction in the target theorem prover. This way one has to prove in the theorem prover that the greatest fixed point does indeed exists before one can use the relation lifting. Further the compiler does only generate those items of the semantics that are well defined. Assume for example a class specification $\mathcal{S}$ that depends on a class specification $\mathcal{S}'$, where $\mathcal{S}'$ contains a type parameter with mixed variance. In this case the compiler does not generate the definition of coalgebra morphism for the signature of $\mathcal{S}$.

The only dangerous point is the semantics of the type constructors for coalgebraic class specifications. It does not make sense to use Rößiger's construction in conjunction with Proposition 4.5.18 to built the final model of a class specification in the target theorem prover. Rößiger's construction is far too complicated for this purpose. Therefore, for the semantics of class specifications, the compiler generates a new type and a few axioms. This can lead to inconsistencies, if the class specification has no model (for loose semantics) or if it does not have a final model (for final semantics).

Therefore the golden rule for using CCSL consistently is

> If S is a class specification processed with loose semantics: Do not proceed until you have constructed a model of S in the target theorem prover.
>
> If S is processed with final semantics, then do not proceed until you have constructed the final model of S.

If this does not give enough security, then one can use the compiler switch `-pedantic`. It causes the compiler to accept only those source files that fulfil the assumptions of Theorem 4.7.4 (see Subsection 4.9.9 for the user interface of the CCSL compiler).

4.8. CCSL and Object Orientation

In this section I investigate the relation of CCSL to the concept object orientation. Chapter 2 of (Meyer, 1997) lists 29 criteria of object orientation. Depending on personal preferences and the interpretation of these criteria one can argue whether CCSL fulfils more criteria than, for instance, Java. The main difference is that CCSL is a specification language. So some of the criteria make no sense at all for CCSL. Consider for instance *dynamic binding* (often called late binding). When a method is called for a specific object, then the method body corresponding to the actual type of the object (in contrast to the static type of the identifier that refers to the object) should be executed. Method declarations in CCSL have no method bodies. Further it is unclear what execution should mean for CCSL. So the term dynamic binding does not apply to CCSL (and not to specification languages in general).

Based on this illustration I consider the question whether CCSL is object oriented as irrelevant. The term object-orientation does apply to software construction systems, it does not apply to a single specification language. So CCSL *is not* an object-oriented specification language. However, I claim that CCSL is a specification language *for* object-oriented programming. A CCSL specification is organised as a series of class and abstract data type specifications. Each class specification contains a number of method declarations, whose behaviour is specified together. This perfectly matches the view of object-oriented programming, where software is organised in classes. However, object orientation consists of more than just the concept of classes.

In the past CCSL has sometimes been criticised for the lack of a particular object-oriented feature. It would indeed be possible to make CCSL more object-oriented in the sense of providing additional syntax for a specific object-oriented feature and including it into the semantics. However, even for key features of object orientation there is no consensus among the object-oriented community on how to do it right. Witness for this are programming languages in the field, for instance C++, Java and Eiffel, which are quite different in their view on object orientation. An attempt to make CCSL more object oriented would necessarily specialise CCSL from a general specification language for object orientation to a specification language for a specific programming language. While it is an interesting challenge in its own to design, for instance, a specification language for Java, the aim of this work was to create a specification language that is independent of a specific programming language. As a result syntax and semantic of CCSL are relatively simple.

In this section I discuss some design choices that have been made for CCSL and compare it with the choices of the programming languages OCAML, Eiffel, C++, and Java. The information about these languages has been taken from (Leroy et al., 2001; Meyer, 1992; Meyer, 1997; Stroustrup, 1997) and (Gosling et al., 1996), respectively. This section is necessarily more informal in style. The arguments in favour of or against a particular decision are often of similar strength, the decision depends then on personal preferences.

The following subsection shed light on the relation of CCSL with inheritance (Subsection 4.8.1), subtyping (Subsection 4.8.2), multiple inheritance (4.8.3), and overriding and dynamic binding (4.8.4).

4.8.1. Inheritance

Inheritance allows one to derive an implementation for a class heir from a class parent without actually copying the source code of parent. In this case the class heir *inherits* from class parent. Equivalently one says that heir is a *descendent* of parent or that parent is an *ancestor* of heir. Inheritance is a key concept of object orientation. For CCSL inheritance is important in two ways. First, it would be nice if a specification for both classes heir and parent has a similar structure. It should consist of a class specification $\mathcal{S}_P$ for the class parent and a class specification $\mathcal{S}_H$ for the class heir such that $\mathcal{S}_H$ is derived from $\mathcal{S}_P$. Second, it is desirable that this derivation at the specification level does not involve textual copying of $\mathcal{S}_P$.

The first point is an abstract property of the involved specifications, it is covered by the notion of subspecification (Definition 4.5.16). The second point is a syntactic feature of CCSL that is independent from the notion of coalgebraic specification. Let me discuss these two points in order.

It is general consensus among the object-oriented programming languages that the heir inherits all instance variables and methods from the parent. Constructors, which are used to create and initialise new objects, are usually not inherited. The rationale is that one cannot expect that a constructor of the parent correctly initialises the heir. These remarks directly apply to Java and C++. In Eiffel constructors are called *creation features.* Creation features are inherited as normal features by the heir (so they can only be used to create new objects of the heir if they are marked as creation features again). All these languages provide syntactic means to invoke a constructor of the parent in a constructor of the heir. The programming language OCAML is a bit different. There one can only specify *initialisers*, which are special expressions evaluated after object creation. In OCAML inheritance propagates initialisers.

For CCSL I adopt the point of view that inheritance should not propagate constructors. Therefore the definitions of subsignature and subspecification (compare Definitions 4.4.3 and 4.5.16) neglect constructor declarations and creation conditions.

The concrete syntax of CCSL contains an inheritance clause with which it is possible to build a subsignature hierarchy without copying. The syntax is as follows.

$$\begin{array}{lll}
\textit{inheritsection} & ::= & \mathtt{INHERIT\ FROM}\ \textit{ancestor}\ \{\!|\ ,\ \textit{ancestor}\ |\!\} \\
\textit{ancestor} & ::= & \textit{identifier}\ [\ \textit{argumentlist}\ ] \\
 & & [\ \mathtt{RENAMING}\ \textit{renaming}\ \{\!|\ \mathtt{AND}\ \textit{renaming}\ |\!\}\ ] \\
\textit{renaming} & ::= & \textit{identifier}\ \mathtt{AS}\ \textit{identifier}
\end{array}$$

An inheritance clause has the following effect: First the type parameters of the ancestor are instantiated with the provided type expressions. Then the instantiated attribute and method declarations are added (disjointly) to the current class together with all method assertions. If an attribute or method identifier of a parent class occurs already in the heir, then it is renamed automatically. The user can rename attributes and methods himself with RENAMING's in order to prevent unintended name clashes. The CCSL compiler takes care that, if a renaming occurs, the inherited method assertions refer to the inherited methods.

4.8.2. Subtyping

A second key concept of object orientation is that one can pass an object o into an environment which expects only a subset of the methods that are available for o. This idea is formalised by enriching the type system with a *subtype relation.* Intuitively a type σ is a *subtype* of τ (alternatively τ is a *supertype* of σ), denoted by $\sigma \leq \tau$, if it is safe to pass an inhabitant of σ to a function that has domain τ. Type systems for object orientation are usually equipped with a subtype relation, see for instance (Pierce and Turner, 1994; Cardelli and Wegner, 1985; Abadi and Cardelli, 1996; Castagna, 1997)

A language has *implicit subtyping* if the programmer is not required to insert a type conversion when he passes an object to an environment that expects a supertype. The languages C++, Eiffel, and Java do have implicit subtyping. In OCAML the types for objects are modelled with parametric polymorphism involving an anonymous type variable (often called the row variable). One of the consequences is that in OCAML one has to use *explicit subtyping*. This means that the programmer has to insert a type conversion into the OCAML source code at each point where an object is passed into a function that expects an object of a different class. One can argue that explicit subtyping has not much to do with subtyping, because a type conversion that converts objects of a subtype σ to a supertype τ can be seen as a function $\sigma \longrightarrow \tau$, so that no subtype relation is required at all. One can also consider implicit subtyping as an additional feature that is provided by the compiler, which automatically inserts a type conversion at every point where types do not match. Indeed, such behaviour is specified for Java (compare §5 in (Gosling et al., 1996)).

CCSL has a semantics in set theory. There, types are represented by sets and implicit subtyping is provided by the subset relation. However, the subset relation is far to restrictive, for instance $M \times N \not\subseteq M$ in general, so CCSL and its semantics cannot provide implicit subtyping.

The subtype relation is often confused with the inheritance relation. In (Cook et al., 1990) it is shown that both relations are independent (see also the discussion in Section 3.1). The programming language OCAML adopts this point of view: It is an easy exercise to write three independent OCAML programs, each containing two classes a and b, that have the following properties. In the first program b inherits from a but a is a subtype of b. In the second program b inherits from a and a and b are not related by subtyping. Finally in the third program b is a subtype of a but neither a inherits from b nor b inherits from a.

In practice software is structured in an inheritance hierarchy and it is often desirable to identify subtyping and inheritance as much as possible. Moreover, understanding a subtype relation is a difficult challenge, whereas understanding an inheritance relation is relatively simple. Therefore many languages identify subtyping and inheritance at the price of loosing (static) type safety. Examples are C++, Java, and Eiffel.

In an object-oriented programming language a class usually gives rise to a type, the type of objects of that class. As a consequence all objects that belong to one class have an uniform structure. In CCSL there is the notion of class specifications and of models of that class specification. One class specification can have different models. The state space of two such models can have different structure. Consider for instance the model of the queue signature in Example 4.4.6. There I used pairs of natural numbers and functions as state space. There are models of this signature in which the state space is a set of functions, in other models it is a set of lists. There is no uniform type for the state space of all models of one class. So for a function that models explicit or implicit subtyping it is not clear what codomain this function should have. For this reason it does not make sense to require that CCSL models implicit or explicit subtyping. The user of CCSL who constructs the models has the choice: He can build the models in a way such that the state spaces are in a subset relation. Alternatively he can provide conversion functions that model explicit subtyping.

Certain conversion functions are always provided through the structural properties of coalgebraic class specifications (Jacobs, 1996a; Jacobs, 1996b; Poll, 2001). Consider a model $\mathcal{M} = \langle X, c, a \rangle$ of a specification $\mathcal{S}$. The subsignature projection $\pi_{\Sigma'}$ that belongs to a subspecification $\mathcal{S}' \leq \mathcal{S}$ yields a coalgebra $\pi_{\Sigma'} \circ c$ that fulfils the assertions of $\mathcal{S}'$, so in a sense, it converts objects that fulfil the specification $\mathcal{S}$ into objects that fulfil the method assertions of $\mathcal{S}'$. Note that it might be impossible to find an algebra a' such that $\langle X, \pi_{\Sigma'} \circ c, a' \rangle$ is a model of $\mathcal{S}'$. This happens because the definition of subspecification and of subsignature put no constraints on constructors and creation conditions. If one uses final semantics, then $\mathsf{coreduce}_{\mathcal{S}'}(\pi_{\Sigma'} \circ c)$ maps objects in X to the canonical model of $\mathcal{S}'$.

4.8.3. Multiple Inheritance

The programming languages Eiffel, C++, and OCAML allow *multiple inheritance*. And so does CCSL. Multiple inheritance means that a given class can inherit from multiple ancestors. In particular it is possible that a given ancestor is inherited twice or more times via different paths. This is called *repeated inheritance*. The question is, if an object of the heir should contain the instance variables (and the methods) of a repeated ancestor once (the repeated ancestor is shared) or several times (the repeated ancestor is not shared). There is no general answer, because there exist examples where sharing has advantages over non-sharing and vice versa. In OCAML common ancestors are never shared (the last copy hides and overrides all previous ones). In C++ the user has the choice via declaring the ancestor class as virtual or not. Eiffel solves the problem via its resolving of name clashes: Features that have the same name are shared (if certain consistency requirements are met), features that have different names are not shared. Java has the simplest answer to this question: it supports only single inheritance.

In CCSL the question is a bit more difficult, because usually the ancestors are parametric in some type variables. So in order to decide whether a given class is inherited twice it is necessary to have an equality relation on types. On the one hand, if CCSL would allow sharing of common ancestors one would need an appropriate equality relation on types. Besides that additional syntax would be necessary to let the user decide whether he or she wants sharing in a particular case or not. On the other hand even if in CCSL repeatedly inherited classes are never shared, an user can easily enforce sharing by an assertion

$$\langle \text{path to first copy} \rangle(x) \quad = \quad \langle \text{path to second copy} \rangle(x)$$

where $\langle$path to $\dots\rangle$ is a suitable combination of subsignature projections.

Under these considerations it seems best to opt for the second alternative: If a class specification is repeatedly inherited in CCSL its method declarations and assertions are included multiple times. Name clashes (one identifier refers ambiguously to more than one declaration) cannot occur, because the semantics of the inherit section is defined via disjoint union. The CCSL compiler automatically renames declarations if otherwise a name clash would occur.

4.8.4. Overriding and Dynamic Binding

Overriding describes the technique to give a new definition for a method that is inherited from an ancestor class. In (Meyer, 1997) Meyer distinguishes *dynamic* and *static binding*. The term *late binding* is a synonym for dynamic binding.

Dynamic binding means that for overridden methods the executed method body is chosen according to the dynamic type of the object. For static binding the type of the variable that holds the object determines the method body that will be executed. Eiffel, Java, and OCAML offer only dynamic binding. In C++ the user has the choice: dynamic binding takes effect if the method is declared as virtual in the parent class and if the object is handled via a reference or a pointer. Otherwise static binding is used.

The first (and more important) question is, how one can model programs in CCSL that exploit dynamic binding. And, secondly, although Meyer considers static binding as "gravest possible crime in object-oriented technology"[26] it is interesting if one can model static binding at the same time. The general answer is that CCSL can model both static and dynamic binding in different ways. In the following I will show several examples to demonstrate how this can be done. All these examples are the result of a long discussion with Bart Jacobs about static and dynamic binding in CCSL.

[26] (Meyer, 1992), page 345

Consider the following OCAML fragment.

```
class parent = object
  method m = 0
end

class heir = object
  inherit parent
  method m = 1
end
```

Class parent contains one method m that returns the integer 0 on every invocation. Class heir inherits from parent and overrides m to return 1. Consider now the method invocation o#m where o is a variable of type parent.[27] What can we derive about the result of o#m? Certainly not that the result is 0, because in a run of the program an instance of heir could have been assigned to o. Assuming that there are no other subtypes of class heir we *can* derive that the result is less than 2. To be more precise we need information about the dynamic type of the object that the variable o holds.

How can a specification for the classes parent and heir look like? It should be possible to reason not only about complete programs but also about program fragments. Thus it would be inadequate to assume that one can derive the dynamic type of every object for every method call in the verification environment. Let $\mathcal{S}_{\mathsf{parent}}$ denote the specification for class parent. There are at least two points of view: First, the *monotone* approach considers the specification $\mathcal{S}_{\mathsf{parent}}$ as a specification of the objects of parent and all its descendents. Second, in the *nonmonotone* approach $\mathcal{S}_{\mathsf{parent}}$ is a specification for the objects of class parent only. Objects of heir do not need to fulfil the method assertions of $\mathcal{S}_{\mathsf{parent}}$.

The monotone approach follows Liskov's substitution principle (Liskov, 1988), which (roughly) says that any context that accepts objects of parent should also accept objects of heir. The monotone approach is further consistent with Eiffel. There the class invariants, the pre- and the postconditions are a specific conjunction of the corresponding properties of the ancestor classes.[28] The nonmonotone approach takes the point of view of a programmer who expects an assertion $\mathsf{o.m} = 0$ in $\mathcal{S}_{\mathsf{parent}}$ because this fits best with the source code of parent. Both approaches are consistent with the semantics of CCSL as described so far. However to force either one, one had to introduce technical complications into the semantics of CCSL. At the point of writing it is not clear if the monotone

[27] OCAML uses o#m instead of o.m to syntactically distinguish method invocation from record selection.

[28] In Eiffel classes can contain logical properties formulated in a special propositional logic. Via a compiler switch the user can enable their evaluation at runtime. If one of the properties is violated it yields an exception (similar to the assert directive in C++). One can specify class invariants (properties that are checked whenever the control flow enters or leaves a feature of that class), preconditions (properties about the arguments of a feature), and postconditions (properties about the return value of a feature).

```
Begin SParent : ClassSpec
  Method
    m : Self -> nat;
  Assertion Selfvar x : Self
    p1 : x.m ≤  1;
End SParent

Begin SHeir : Classspec
  Inherit From SParent
  Assertion Selfvar x : Self
    h1 : x.m = 1;
End SHeir
```

Figure 4.17.: The monotone approach to model dynamic binding.

approach is superior to the nonmonotone or vice versa. It seems that the decision, which approach to prefer, depends very much on the concrete verification problem. Moreover both approaches are equivalent in the sense that if the semantics of CCSL would be monotone, then it would be possible to write specification that mimic nonmonotone semantics and vice versa. As a conclusion from these various considerations it seems best to leave the semantics of CCSL as simple as possible and let the user decide. In the following I describe how one can model the monotone and the nonmonotone style of specification in CCSL.

In the monotone style one adds assertions to further restrict the inherited methods. The resulting specifications are in Figure 4.17. The specifications SParent and SHeir in Figure 4.17 are both consistent. The only problem that remains is that, in case we know that an object of type parent is assigned to a variable o then, we cannot derive that o#m = 0. To fix this it is necessary to incorporate the notion of dynamic type into the specification. The simplest way to do this is to assume that the ground signature contains a type of sufficient cardinality that models the dynamic types of the objects. For simplicity I use the natural numbers here, and let 0 be the dynamic type of parent and 1 be the dynamic type of heir. The modified specification that takes dynamic type information into account is in Figure 4.18.

Now we can derive o#m = 0 provided we have information about the dynamic type of o. The subsignature projection is linked to dynamic binding because it does not change the value of dynamic_type. In terms of Java it is a widening reference conversion (§5.1.4 in (Gosling et al., 1996)).

The specification in Figure 4.18 does not model static binding. One can easily fix this by adding a method declaration static_parent : Self ⇒ Self with an additional assertion dynamic_type(static_parent(x)) = 0, where x is a variable of type Self.

```
Begin SParent : ClassSpec
  Method
    dynamic_type : Self -> nat;
    m : Self -> nat;
  Assertion Selfvar x : Self
    p1 : x.m ≤  1;
    p2 : dynamic_type(x) = 0 Implies x.m = 0;

  Constructor
    new_parent : Self;
  Creation
    p3 : dynamic_type(new_parent) = 0
End SParent

Begin SHeir : ClassSpec
  Inherit From SParent
  Assertion Selfvar x : Self
    h1 : dynamic_type(x) = 1 Implies x.m = 1;

  Constructor
    new_heir : Self;
  Creation
    h2 : dynamic_type(new_heir) = 1
End SHeir
```

Figure 4.18.: The monotone approach to model dynamic binding taking dynamic type information into account.

The modelling of the nonmonotone approach follows ideas from the Java branch in the LOOP project, see (Huisman and Jacobs, 2000). An example specification for the nonmonotone approach is in Figure 4.19. Note that the specification SHeir is consistent because during inheritance the method declaration m of SParent is renamed, say to parent_m, and the inherited assertion p_1 refers to parent_m. The important thing to note is that now the subsignature projection corresponds to static binding. The problem in this approach lies in the modelling of dynamic binding.

Let me fix some notation to explain how this works. Let $\mathcal{S}_P = \langle \Sigma_P, \{\mathsf{p}_1\}, \emptyset \rangle$ be the coalgebraic specification for SParent and let $\mathcal{S}_H = \langle \Sigma_H, \{\mathsf{p}'_1, \mathsf{h}_1\}, \emptyset \rangle$ be the specification for SHeir. A model for $\mathcal{S}_H$ consists of a state space X together with a coalgebra $c : X \longrightarrow \mathbb{N} \times \mathbb{N}$ where $\pi_1 \circ c$ interprets the method parent_m and $\pi_2 \circ c$ interprets m. The subsignature projection maps c to $\pi_1 \circ c$, which is a model for $\mathcal{S}_P$. The trick in

```
Begin SParent : ClassSpec
  Method
    m : Self -> nat;
  Assertion Selfvar x : Self
    p1 : x.m = 0;
End SParent

Begin SHeir : ClassSpec
  Inherit From SParent
  Method
    m : Self -> nat;
  Assertion Selfvar x : Self
    h1 : x.m = 1;
End SHeir
```

Figure 4.19.: Example for the nonmonotone approach to model dynamic binding.

getting dynamic binding to work lays in a suitable rearrangement of the methods in the coalgebra. In this simple example we see that $\pi_2 \circ c$ is a (signature) model for Σ_P in which the interpretation of m provably equals 1. Note that $\pi_2 \circ c$ does not fulfil the assertion p_1. With sophisticated rearrangements one can model upcasts that bind some methods statically and some dynamically. This can be used to model C++, where dynamic binding applies only to methods that are declared as virtual. In (Huisman and Jacobs, 2000) this technique is used to model the widening reference conversion of Java. The special feature of Java is that this conversion uses dynamic binding for the methods and static binding for the fields (instance variables).

4.9. Miscellaneous

This section explains those parts of CCSL that do not fit into one of the previous sections. The first subsection describes the structure of input files and what the compiler generates. The following subsections are on the include directive, on lifting requests, importings, infix operators, (qualified) identifiers, anonymous ground signatures, the prelude, the user interface of the CCSL compiler, and the implementation and internal structure of the current implementation.

4.9.1. Input and Output Files

A complete CCSL specification consists of a sequence of ground signature extensions, class specifications, and abstract data type specifications. Such a sequence can be spread over several files by using the include directive (see the following Subsection). In the following grammar rules the meta symbol *file* stands for a complete CCSL specification (and not for the contents of exactly one file).

$$
\begin{array}{lcl}
\mathit{file} & ::= & \{\!|\ \mathit{declaration}\ |\!\}\ \texttt{EOF} \\[1ex]
\mathit{declaration} & ::= & \mathit{classspec} \\
 & | & \mathit{adtspec} \\
 & | & \mathit{groundsignature} \\
 & | & \mathit{typedef} \\
 & | & \mathit{groundtermdef}
\end{array}
$$

The meta symbol *typedef* is also allowed outside of ground signatures. Together with the meta symbol *groundtermdef* it provides a lightweight syntax for ground signature extensions, see 4.9.7 below.

The theorem provers ISABELLE and PVS organise their input material in *theories*. Each theory can depend on a number of other theories and contains axioms, type and constant declarations, and proof goals. For PVS it is important that all the material in one theory depends on all type parameters of the theory. Therefore PVS theories tend to be rather short. For PVS one file can contain several theories. In ISABELLE there is no problem with the type parameters. However, an ISABELLE file might contain only one theory.

In this setting the CCSL compiler behaves as follows. For each specification or ground signature ⟨`spec`⟩ in the input file it generates a number of different *internal* theories. What theories are precisely generated depends on the properties of the input and on the target theorem prover. For PVS the compiler dumps one internal theory into one PVS theory. All theories that belong to the specification ⟨`spec`⟩ are put into one file ⟨`spec`⟩`_basic.pvs`. For ISABELLE the compiler combines all internal theories into one ISABELLE theory ⟨`spec`⟩`_basic` and prints it into the file ⟨`spec`⟩`_basic.thy`.

If a class specification contains a theorem section then the formulas there are translated into a theory ⟨`spec`⟩`Theorem` and written into a separate file.

For a ground signature ⟨`gsig`⟩ that defines types or constants version 2.2 of the CCSL compiler generates one theory ⟨`gsig`⟩`Definition` and possibly several theories ⟨`gsig`⟩`Definition`*n*, where *n* stands for a generated sequence number. The reason for separating the material of one ground signature into several theories lays in the different treatment of type parameters in PVS and CCSL. The compiler generates no output for ground signatures that contain only declarations (i.e., no defining equations).

For an abstract data type specification ⟨`adt`⟩ the CCSL compiler can generate the following theories.

name of theory	contents
⟨`adt`⟩	data type declaration
⟨`adt`⟩`Util`	reduce, accessors, recognisers
⟨`adt`⟩`Map`	map combinator
⟨`adt`⟩`Every`	(full) predicate lifting
⟨`adt`⟩`RelLift`	(full) relation lifting

The theory ⟨`adt`⟩`Util` is only generated for ISABELLE. The theories for the map combinator and for the liftings are generated if these notions exist for the adt in question and if the target theorem prover does not provide them.

The theories that can be generated for a class specification are displayed in the Tables 4.20 and 4.21. Again, these theories are only generated, if there contents is defined for the actual class specification. For instance for class specification for which loose semantics is requested the theories for the final model and for the map combinator are not generated.

The CCSL compiler generates a fair amount of theorems. Unfortunately, it generates only very few proofs. For PVS this is no problem. For ISABELLE the compiler generates sorry[29] proofs.

4.9.2. Include Directive

The CCSL compiler supports a C-preprocessor like include directive:

include ::= `#include "`*string*`"`

The string must be the name of a file, which is literally substituted for the include directive. The include directive is handled by the lexer, it can appear at any place in the input.

4.9.3. Lifting Requests

During type checking the CCSL compiler determines all uses of behavioural equality, derives the types and generates appropriate liftings. However, sometimes an user wishes to use behavioural equality for types that do not occur in the specification. The CCSL compiler supports these users via lifting requests. A lifting request consists of a name and a type expression. The compiler generates the relation lifting for this type and adds a declaration with the given name in the generated files.

requestsection ::= `REQUEST` *request* ⦃ ; *request* ⦄

request ::= *identifier* : *type*

[29] The ISAR command sorry does a fake proof pretending to solve the pending proof goal without further ado (Wenzel, 2002).

name of theory	contents
⟨`class`⟩`Interface`	signature declaration
⟨`class`⟩`Method_Id`	tags for method wise modal operators
⟨`class`⟩`MethodPredicateLifting`	(method wise) predicate lifting, method-wise invariants
⟨`class`⟩`MethodInvariantRewrite`	utility lemmas for invariants
⟨`class`⟩`MethodInvariantInherit`	link methodwise invariants with super classes
⟨`class`⟩`Box`	(method wise) modal operators
⟨`class`⟩`BoxInherit`	link modal operators with super classes
⟨`class`⟩`Bisimilarity`	relation lifting and bisimulations
⟨`class`⟩`BisimilarityRewrite`	utility lemmas for bisimulations
⟨`class`⟩`PublicBisimilarityRewrite`	utility lemmas for bisimulations with respect to the public subsignature
⟨`class`⟩`BisimilarityEquivalence`	bisimulation on one model, bisimilarity
⟨`class`⟩`BisimilarityEqRewrite`	utility lemmas for bisimilarity
⟨`class`⟩`PublicBisimilarityEqRewrite`	utility lemmas for bisimilarity with respect to the public subsignature
⟨`class`⟩`ReqObsEq`	additional liftings
⟨`class`⟩`Morphism`	definition of ⟨`class`⟩–coalgebra morphisms
⟨`class`⟩`MorphismRewrite`	utility lemmas for morphisms
⟨`class`⟩`Semantics`	semantics of the specification
⟨`class`⟩`Basic`	utility lemmas for assertions and creation conditions
⟨`class`⟩`FullInvariant`	full predicate lifting and every combinator
⟨`class`⟩`FullBisimulation`	full relation lifting and relevery combinator
⟨`class`⟩`Finality`	properties of the final ⟨`class`⟩–coalgebra, coreduce
⟨`class`⟩`FinalityBisim`	Bisimilarity on the final model
⟨`class`⟩`Final`	axiomatic final model
⟨`class`⟩`FinalProp`	axiom for final model

Table 4.20.: Generated theories for a class specification ⟨`class`⟩, Part I

name of theory	contents
⟨`class`⟩`MapStruct`	coalgebra structure for map combinator
⟨`class`⟩`Map`	map combinator
⟨`class`⟩`Loose`	axiomatic loose model
⟨`class`⟩	top level import theory for ⟨`class`⟩
⟨`class`⟩`Theorem`	translated theorem section

Table 4.21.: Generated theories for a class specification ⟨`class`⟩, Part II

4.9.4. Importings

In PVS and in ISABELLE there must be a strict hierarchy between all theories. One can only use the identifiers that are declared in the current theory or in one of the theories on which the current theory depends.

Therefore it is necessary that the CCSL compiler generates the right dependencies between the theories in its generated output. During parsing the CCSL compiler collects all type constructors that are used in each specification and generates the right dependencies. For ground signatures that declare nonstandard material it is necessary to inform the CCSL compiler where the material in the ground signature is defined. For special applications it is sometimes necessary to adapt the automatically inferred dependency relation. All this is done via the importing clause (in PVS the dependency between theories is given by importing statements). Importings can occur at the beginning of ground signatures or class specifications, or in a section for assertions or creation conditions.

$$\textit{importing} \quad ::= \quad \texttt{IMPORTING}\ \textit{identifier}\ [\ \textit{argumentlist}\ ]$$

For PVS it is sometimes preferable to instantiate parametric theories in importing statements. Therefore it is possible to provide an argument list in the CCSL importing clause. For ISABELLE the arguments are suppressed.

4.9.5. Infix Operators

CCSL permits the declaration of infix operators in ground signatures to allow expressions like $3 + 4$ in assertions. The infix operators of CCSL are very similar to the ones of OCAML (Leroy et al., 2001) and use the same implementation technique. Infix operators can be several characters long. They are sequences of the following characters

```
! $ & * + - . / \ : < = > ? @ ^ | ~ #
```

where the first character is one of

```
$ & * + - / \ < = > @ ^ | ~ #
```

Infix operators are grouped into precedence levels according to their first characters. Associativity is fixed and depends also on the first characters. Operators starting with `**` have the highest precedence. These operators are right associative. On the next precedence level are the operators which have `*`, `/` or `\` as first character, followed by those with `+` or `-`, followed by the operators starting with `@`, `^`, or `#`. All these operators are left associative. On the least precedence level are the operators starting with `=`, `~`, `<`, `>`, `|`, `&`, or `$`. They are non-associative.

Infix operators must be declared as functions taking two arguments, so their type must have a structure either like $(\tau \times \sigma) \Rightarrow \rho$ or like $\tau \Rightarrow \sigma \Rightarrow \rho$. If an infix operator is surrounded by a pair of parenthesis it becomes a (prefix) function symbol. In the declaration in the ground signature the parenthesis are also required.

Two infix operators are predefined: `=` for equality and $\sim$ for behavioural equality.

4.9.6. Identifiers and Qualified Identifiers

Identifiers in CCSL are sequences of letters, digits, the underscore, and the question mark. Identifiers must begin with a letter. The list of reserved words is in the Appendix B on page 284.

Let me use the term *specification* in this subsection to denote a class specification, a ground signature, or an abstract data type specification that occurs in the CCSL input. Any specification defines certain items for the specifications that follow, as explained in Section 4.7. One can use a *qualified identifier* to refer to one of these items, even if the identifier is hidden by another declaration. Qualified identifiers can occur at the expression level in assertions (denoting constants or functions) or in type expressions (denoting types from a ground signature). Their syntax is as follows.

$$\begin{array}{lcl} \textit{qualifiedid} & ::= & \textit{idorinfix} \\ & | & \textit{identifier} \; [\; \textit{argumentlist} \;] \; \underline{::} \; \textit{idorinfix} \\ \textit{idorinfix} & ::= & \underline{(}\; \textit{infix\_operator} \; \underline{)} \\ & | & \textit{identifier} \end{array}$$

The meta symbol *idorinfix* (whose definition is repeated here for convenience) stands for an unqualified identifier, which may be an infix operator in parenthesis. A qualified identifier consists of a specification identifier, an optional argument list, and an unqualified identifier. If the specification declares type parameters the argument list must be present.

4.9.7. Anonymous Ground Signatures

It is possible to define or declare type constructors and constants outside of ground signatures with the keywords `TYPE` and `CONSTANT`. The concrete syntax is the same as inside ground signatures. For convenience I repeat the relevant meta symbols from the grammar:

typedef ::= `TYPE` *identifier* [*parameterlist*] [`=` *type*]
groundtermdef ::= `CONSTANT` *termdef* [`;`]
termdef ::= *idorinfix* [*parameterlist*] `:` *type* [*formula*]

The CCSL compiler combines any sequence of such declarations into an anonymous ground signature.

4.9.8. The Prelude

Before processing the actual input file the CCSL compiler parses a string that is hard wired into the compiler: *the CCSL prelude*. The prelude extends the ground signature to contain some basic types and constants. The prelude of the compiler version 2.2 is displayed in Figure 4.22. The ground signatures EmptySig and EmptyFunSig belong only to the prelude, if the target theorem prover is PVS. For ISABELLE the data type of lists has constructors Nil and Cons to match ISABELLE's definition. The compiler is clever enough to avoid the repetition of the list data type in the target theorem prover.

4.9.9. User Interface

The CCSL compiler is a command line tool in the Unix tradition. Besides command line switches it expects its source files on the command line and outputs into files in the current directory (unless option `-d` is present). Here is a selection of the command line switches for version 2.2 (for a complete listing see the reference manual (Tews, 2002a) or the manual page).

`-fixedpointlib path` Set the location of the PVS fixed point library. The `path` of the fixed point library appears in the generated output. It must point to the correct location, otherwise type checking in PVS fails. The default path is set during installation.

`-d dir` Place all generated files in directory `dir`

`-pvs` Set the target theorem prover to PVS. This is the default.

`-isa` Set the target theorem prover to ISABELLE/HOL in the syntax of new style Isar theories (Wenzel, 2002). Of a sequence of `-pvs` and `-isa` options the last one takes effect.

`-nattype type` Set the type name of natural numbers to `type`. Defaults to `nat`. More precisely, the type checker uses type `type` for all natural number constants (consisting only of digits) in the source. This option is necessary to prevent type

```
Begin EmptySig : GroundSignature
   Importing EmptyTypeDef
   Type EmptyType
End EmptySig

Begin EmptyFunSig [A : Type]: GroundSignature
   Importing EmptyFun[A]
   Constant
      empty_fun : [EmptyType -> A];
End EmptyFunSig

Begin list[ X : Type ] : Adt
   Constructor
      null : Carrier;
      cons( car, cdr ) : [X, Carrier] -> Carrier
End list

Begin Lift[ X : Type ] : Adt
   Constructor
      bot : Carrier;
      up( down ) : X -> Carrier
End Lift

Begin Coproduct[ X : Type, Y : Type ] : Adt
   Constructor
      in1(out1) : X -> Carrier;
      in2(out2) : Y -> Carrier;
End Coproduct

Begin Unit : Adt
   Constructor
      unit : Carrier;
End Unit
```

Figure 4.22.: The CCSL prelude

checking errors if you use natural number constants in combination with a type different from `nat` (for instance `int`).

Note that `type` must be a valid type at each occurrence of a natural number constant in the source. So you probably need to add a ground type declaration for `type` at the start of the specification.

`-batch` Generate a batch processing file. The precise behaviour depends on the output mode. For PVS the compiler generates a file pvs-batch.el containing Emacs lisp code. For ISABELLE the file is called ROOT.ML and contains SML code.

`-class spec` Only generate output for specification `spec`. Repeat this option to get output for several classes.

`-dependent-assertions` Normally the semantics of an assertion is a predicate on the state space that is independent from all other assertions. With this option each assertions has the preceding assertion as assumption. This does not change the semantics of a class specification. However, it makes it possible to discharge type-check conditions (TCC's) with the help of previous assertions.

`-pedantic` Enforce all assumptions of Theorem 4.7.4 except the consistency requirement for class specifications. To ensure invariance with respect to behavioural equality the compiler performs a syntactic check according to the Propositions 4.5.6 and 4.5.11. This check is relaxed in the following two cases:

- Polymorphic constants are recognised as behaviourally invariant if they are instantiated with a constant type.
- Constructors of a class specification are allowed in the creation assertions of that class specification. This rests on the construction in the proof of Proposition 4.5.18.

`-expert` Turn on expert mode. This turns a number of errors into warnings. As a result the compiler might generate inconsistent output.

`-no-opt` Turn off formula optimisation. Normally the CCSL compiler performs several optimisations before printing formulae and expressions. (The compiler uses simple equivalences for optimisation like $\top \wedge p = p$ but assumes also that all ground signature extensions are proper.)

`--help` Print usage information.

`-v` Verbose. Print some messages about compilation progress.

4.9.10. Implementation

The CCSL compiler is implemented in the programming language OCAML (Leroy et al., 2001) using standard compiler construction techniques (see for instance (Aho et al., 1986)). It is organised in several passes. Version 2.2 consists of about 40.000 lines in about 100 files. OCAML is a strongly typed functional programming languages similar to SML (Milner et al., 1991). It contains extensions for an imperative programming style (references, while loops) and also for an object-oriented style.

OCAML was chosen for the following reasons: The transformation of CCSL into higher-order logic requires a lot of symbolic manipulations. This can most easily be programmed with abstract data types and pattern matching. OCAML integrates well with a standard Unix environment and with Emacs. The OCAML distribution contains, besides the compiler, OCAML versions of Lex and Yacc, the replay debugger that allows one to run programs *backwards*, and an extensive library. The compiler itself is small and produces fast code. One of the disadvantages of OCAML is that the object-oriented constructs integrate only purely with the functional part of the programming language. It is for instance not possible to define a set of mutually recursive types such that some of the types are classes and the other are abstract data types. To define such a set of types one has to use the special construction of object types, which makes the whole code very complex. A second problem with OCAML is that it does not allow one to specialise the result type of methods during inheritance.

Some of the intermediate data structures of the CCSL compiler are defined as classes and some as variant types. The three important classes are called `iface`, `member`, and `theory_body`. The internal representations of abstract data type specifications, class specifications, and ground signature extensions are derived from the class `iface` by inheritance. The classes for attributes, methods, constructors, and ground signature constants are derived from the class `member`. The class `theory_body` is the data structure to capture the output that the CCSL compiler generates. There is one specific class that inherits from `theory_body` for every theory in the output.

The variant types formalise the internal representation of types, formulae and expressions.[30] All these types are mutually dependent, for instance formulae are expressions (of boolean type), expressions contain types, type constructors that occur in types can stem from class specifications, and, finally, a class specification contains formulae. This dependency suggests to define all involved types in one mutual recursion. However this is not feasible for several reasons. The most important is the absence of method specialisation: The class `iface` contains a method `get_members` that returns a list of `member`'s. Using inheritance one wants to define the class `ccsl_iface` that overrides `get_methods` such that it returns now a list of `ccsl_member`'s. However, the OCAML type checker requires that the overriding method has the same type as the overridden method.

One solution to this problem (which has been adopted in the CCSL compiler) is to

[30] Although higher-order logic does not distinguish between formulae and expression, it is conceptually easier to use different types for formulae and expressions internally.

write a class pre_iface that is polymorphic in a type variable α (possibly constraining α to be a subtype of member). The method get_methods in pre_iface returns a list of α's. The desired iface class can be obtained by instantiating α with member. By inheritance one can derive a polymorphic class ccsl_pre_iface. The class ccsl_iface is obtained from ccsl_pre_iface by instantiating it with ccsl_member. Now the method get_members in ccsl_iface has the desired type. Note that in OCAML the class ccsl_iface *is not* a subtype of class iface.

For the CCSL compiler we introduced several type parameters to break the (formal) dependency between the type definitions. All the variant types are polymorphic in two type variables, one is instantiated with the internal type for class specifications, the other is instantiated with the internal representation of methods (and attributes). The class pre_iface takes three type parameters. The first will be instantiated with an instance of class member, the second with an instance of theory_body, and the third type variable is instantiated with the class iface itself. For instance the file ccsl_classtype.ml contains the following code.

```
class type ccsl_iface_type
  = [ccsl_member_type,
      (ccsl_iface_type, ccsl_member_type) ccsl_pre_theory_body_type,
      ccsl_iface_type] ccsl_pre_iface_type

and ccsl_member_type =
    [ccsl_iface_type, ccsl_member_type] ccsl_pre_member_type
```

This code fragment defines suitable instantiations of iface and member as abbreviations ccsl_iface_type and ccsl_member_type, respectively. The instantiations are in brackets. The second argument for ccsl_pre_iface_type must be instantiated as well. For some obscure reason one has to group these instantiations with parenthesis (instead of brackets). Note that both abbreviations are (mutually) recursive. This is no problem in OCAML.

The CCSL compiler consists of the following passes

Lexing & Parsing The lexer and the parser are generated by ocamllex and ocamlyacc, respectively. Theses are the OCAML variants of LEX and YACC. Keywords are recognised with a hash table that sits in between the lexerer and the parser. The contents of this keyword hash table is generated from the YACC source by a home grown tool, which has been inspired by GPERF (Schmidt, 1990).

The parser resolves all type identifiers. Identifiers for variables, methods and constructors are resolved later.

The result of the parser is an abstract syntax tree. All following passes work on this syntax tree and add information by destructive updates. This syntax tree contains information about source code locations such that later passes can generate exact error messages.

Update Methods Scan class signatures for attributes and generate update methods.

Inheritance Resolve inheritance in class specifications. Lookup ancestors, instantiate them, perform method renaming, check for name clashes, and update the symbol table of the heir. When this pass is completed all inherited methods can be found via the symbol table of the heir.

Update Assertions Generate update assertions. Take inherited attributes and inherited update methods into account.

Variance Compute variances for all type parameters. Classify interface functors for data types and classes according to the hierarchy of functors in this thesis.

Features Depending on the type constructors (and their instantiation) that are used in class and data type signatures this pass determines which parts of the general semantics described in Section 4.7 are defined for every specification.

Special Class Members Definition of the special class member `coreduce`, and the recognizers for invariants (⟨`class`⟩`_class_invariant?`),
bisimulations (⟨`class`⟩`_class_bisimulation?`),
and morphisms (⟨`class`⟩`_class_morphism?`).

Resolution Resolution of variables, methods and constructors.

Type Checking Type check all assertions. The type checker is based on unification of types. It temporarily inserts (internal) free type variables into the abstract syntax tree. Their solutions are determined with unification.

Theory Generation Generate all theories in an internal version of higher-order logic.

Pretty Printing Dump the generated theories in the syntax of the target theorem prover.

This description shows clearly that the design of the CCSL compiler is not optimised for efficiency. However, efficiency has never been a problem: An input file of a few hundred lines is processed in less than a second.

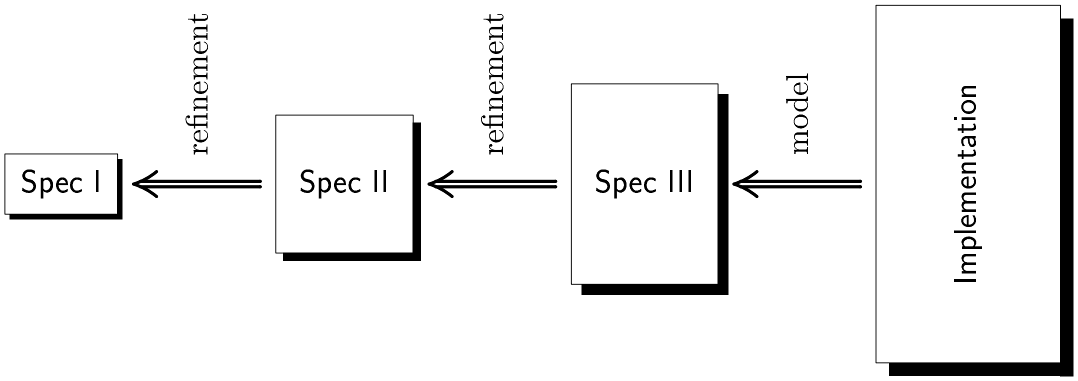
4.10. Applications of CCSL

CCSL has been used in the past in a number of larger and smaller case studies. A major point in these case studies has been the construction of coalgebraic refinements. The first subsection describes the notion of coalgebraic refinement that is used in the LOOP project at the moment and how one can prove such refinements with CCSL. In the second subsection I review some CCSL case studies. The third subsection applies CCSL to an UML/OCL example.

4.10.1. Proving Coalgebraic Refinement

Refinement is a relation between specifications. It links a specification that is considered to be more 'abstract' with a specification that is considered to be more 'concrete'. The intuition is as follows: A concrete specification $\mathcal{S}_C$ *refines* an abstract specification $\mathcal{S}_A$, if all models of $\mathcal{S}_C$ can be transferred into models of $\mathcal{S}_A$. Refinement could also be paraphrased as relative model construction: If $\mathcal{S}_C$ refines $\mathcal{S}_A$ then one can build a model of $\mathcal{S}_A$ by assuming an arbitrary model of $\mathcal{S}_C$. Typically refinement involves a translation of signatures: The operations of the signature of $\mathcal{S}_A$ must be expressed with the operations available in $\mathcal{S}_C$.

Refinement is an important notion in software verification. Instead of relating the implementation directly with the specification one often uses several refinement steps, as depicted below.



As the size of the boxes indicates there is a chain of increasingly complex and increasingly more concrete specifications. The last specification in the chain, Spec III, sufficiently resembles the implementation. So it is feasible to prove that the implementation is a model of Spec III. The specifications are related by refinements. If the chosen notion of refinement is compositional (i.e., if the refinement relation is transitive) it follows that the implementation is also a model of the specification Spec I.

(Wirsing, 1990) describes refinement in the context of algebraic specification, but see also (Back and von Wright, 1998) for program refinement. For coalgebraic specification refinement was first studied in (Jacobs, 1997a) and in (Jacobs, 1997b). Poll defines in (Poll, 2001) a notion of behavioural subtyping between coalgebras and discusses its relation with coalgebraic refinement. The experience with constructing refinements of CCSL specifications in the theorem prover PVS showed that Jacobs' original notion of coalgebraic refinement is not general enough. A number of generalisations finally lead to two notions of refinement: *assertional refinement* and *behavioural refinement.* Both notions and the need for the generalisations are discussed in detail in the joint work (Jacobs and Tews, 2001).

Assertional refinement requires that the assertions of the abstract specification should hold for each translated model of the concrete specification. This implies that for assertional refinement the translation of signatures must cover the complete signature of the abstract specification (because every method of the abstract signature can occur in the assertions). Sometimes this complete coverage is inappropriate. For instance, if the abstract signature contains private methods then one might want to construct a refinement for the public methods only.

In behavioural refinement one requires that each translated model of the concrete specification should be *behaviourally equal* to some abstract model. The behavioural equality can be taken with respect to a subsignature of the abstract specification, for instance to hide the private methods.

In the following I present the definitions from (Jacobs and Tews, 2001) in the formal context of this thesis and explain how one can prove refinements of CCSL class specifications with the theorem prover PVS. Recall from the Definitions 4.4.1 and 4.5.14 (on page 150 and 178, respectively) that a class specification $\mathcal{S}$ is a triple $\langle \Sigma, \mathcal{A}_M, \mathcal{A}_C \rangle$, where $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ is the signature, consisting of the method declarations Σ_M and the constructor declarations Σ_C, and $\mathcal{A}_M$ and $\mathcal{A}_C$ are sets of method and constructor assertions, respectively.

Definition 4.10.1 (Assertional Refinement) Let S_C be a concrete coalgebraic class specification over the signature Σ_C with n type parameters and let S_A be an abstract coalgebraic class specification over the signature Σ_A with m type parameters $\alpha_1, \ldots, \alpha_m$.

1. A *parameter translation* from Σ_A to Σ_C is a n-tuple of types $(\tau_1, \ldots, \tau_n)$ such that every τ_i contains only the type variables $\alpha_1, \ldots, \alpha_n$, that is

   $$\alpha_1 : \mathsf{Type}, \ldots, \alpha_m : \mathsf{Type} \quad \vdash \quad \tau_i : \mathsf{Type}$$

   can be derived.

2. Let $(\tau_1, \ldots, \tau_n)$ be a parameter translation from Σ_A to Σ_C. A fixed interpretation $U_1, \ldots, U_m$ of the type parameters $\alpha_1, \ldots, \alpha_m$ induces an interpretation $[\![\tau_i]\!]_{U_1,\ldots,U_m}$ of the types τ_i. A *translation map* (from Σ_C to Σ_A with respect to $(\tau_1, \ldots, \tau_n)$) is a family of mappings $(\phi_{U_1,\ldots,U_m})$ such that for an interpretation $U_1, \ldots, U_m$ of the type parameters $\alpha_1, \ldots, \alpha_m$ the map $\phi_{U_1,\ldots,U_m}$ assigns to every model $M = \langle X, c, a \rangle_{[\![\tau_1]\!],\ldots,[\![\tau_m]\!]}$ of the specification $\mathcal{S}_C$ a signature model $\phi(M) = \langle X', c', a' \rangle_{U_1,\ldots,U_m}$ of Σ_A such that $X' \subseteq X$.

3. A translation map ϕ is an *assertional refinement*, if $\phi(M)$ is a model of S_A for all models M of S_C.

The notion of parameter translation is not present in (Jacobs and Tews, 2001), there we discuss only refinements between class specification without type parameters. The parameter translation deals with the (rare) situation where the number of type parameters of the abstract and the concrete specification differ. In most cases the parameter translation is the identity, that is $\tau_i = \alpha_i$.

The main restriction in the preceding definition of assertional refinement is that the state space of the translated model $\phi(M)$ is a subset of the state space X of the original model. The requirement that X' must only be a subset of X accounts for the fact that in a refinement one might want to exclude certain (unreachable) states from M.

Definition 4.10.2 (Behavioural Refinement)

1. Let $\mathcal{S}$ be a coalgebraic class specification over signature $\Sigma = \langle \Sigma_M, \Sigma_C \rangle$ and let $\Sigma' = \langle \Sigma'_M, \Sigma'_C \rangle$ be a subsignature of Σ with the same set of constructors: $\Sigma_C = \Sigma'_C$. A *behavioural model* (of $\mathcal{S}$ with respect to Σ') is a model $\langle Y, d, b \rangle$ of Σ' such that there exists a model $\langle X, c, a \rangle$ of $\mathcal{S}$ with $\mathrm{Rel}(\llbracket \sigma_\Sigma \rrbracket)({}_c\!\underline{\leftrightarrow}_d)(a, b)$, where σ_Σ is the combined constructor type of Σ (see page 151).

2. Assume a concrete specification, consisting of a coalgebraic class specification $\mathcal{S}_C$ over the signature $\Sigma_C = \langle \Sigma_{CM}, \Sigma_{CC} \rangle$ with n type parameters and a subsignature $\Sigma'_C = \langle \Sigma'_{CM}, \Sigma'_{CC} \rangle$ such that $\Sigma_{CC} = \Sigma'_{CC}$. Let $\mathcal{S}_A$ be an abstract coalgebraic specification over a signature $\Sigma_A = \langle \Sigma_{AM}, \Sigma_{AC} \rangle$ with a subsignature $\Sigma'_A = \langle \Sigma'_{AM}, \Sigma'_{AC} \rangle$. A translation map ϕ from Σ'_C to Σ'_A is called a *behavioural refinement* from $\langle \mathcal{S}_C, \Sigma'_C \rangle$ to $\langle \mathcal{S}_A, \Sigma'_A \rangle$ if ϕ maps behavioural models of $\mathcal{S}_C$ to behavioural models of $\mathcal{S}_A$.

Both notions of refinements are compositional. If one considers behavioural refinements from $\langle \mathcal{S}_C, \Sigma_C \rangle$ to $\langle \mathcal{S}_A, \Sigma_A \rangle$ (i.e., the case where no methods are hidden) then, under certain (reasonable) assumptions on the assertions of the abstract specification, one can prove that assertional refinement coincides with behavioural refinement. See (Jacobs and Tews, 2001) for details.

In the remainder of this subsection I construct an assertional refinement for the queue specification of Figure 4.11 (on page 184). The refinement is based on the idea that lists form queues, if one appends new elements at the end. The refining specification ListQueue is in Figure 4.23. The complete source code of the refinement and the proofs are available in the world wide web, see Appendix A.

In Figure 4.23 the ground signature ListOp introduces the function append for the concatenation of lists, which is predefined in PVS. The class specification ListQueue has only two methods contents and set_contents, where the latter is automatically generated by the CCSL compiler as an update method for the attribute contents (see Subsection 4.4.2). The method set_contents has the following type

```
set_contents : [Self, list[A]] -> Self
```

```
Begin ListOp[ A : Type ] : GroundSignature
  Constant
    append : [list[A], list[A] -> list[A]];
End ListOp

Begin ListQueue[ A : Type ] : ClassSpec
  Attribute
    contents : Self -> list[A];

  Defining
    put : [Self, A] -> Self
    put(x,a) = set_contents(x, append(contents(x), cons(a,null)));

    top : Self -> Lift[[A, Self]]
    top x = Cases contents x of
                null : bot,
                cons(a,rest) : up(a, set_contents(x,rest))
              EndCases;

  Constructor
    l_new : Self

  Creation
    new_empty : contents(l_new) = null;
End ListQueue
```

Figure 4.23.: A refinement of queues in CCSL

Further the CCSL compiler generates the following assertion.

```
contents_set_contents : Forall( l : list[A] ) : contents(set_contents(x, l)) = l
```

So models of ListQueue are records with one field of type list[A].

An assertional refinement of the queue specification consists of three items: first, a parameter translation, second, a translation function that maps models of ListQueue to models of the queue signature, and, third, a proof that the result of the translation function is a model of Queue. A parameter translation is not necessary in this example, so I choose the identity translation. This means that models of ListQueue[A] get translated into models of Queue[A].

For the second ingredient of an assertional refinement I need an interpretation of the methods top and put for an arbitrary model of ListQueue. Because it is so obvious how to do that I defined these two methods in the ListQueue specification as definitional extensions.

```
QueueRefine[ X, A : Type] : Theory
Begin
  Importing ListQueueBasic[X,A]
  c : Var (ListQueueAssert?)

  Importing QueueBasic[X,A]

  abs_c(c) : QueueSignature[X,A] =
    (# top := top(c),
       put := put(c)    #)

  abs_new(c)(z : (ListQueueCreate?(c))) : QueueConstructors[X,A] =
      (# new := l_new(z) #)

  model : Proposition Forall(z : (ListQueueCreate?(c))) :
      QueueModel?(abs_c(c), abs_new(c)(z))
End QueueRefine
```

Figure 4.24.: The theory ListQueue containing the refinement proof

It remains to prove that the queue assertions hold. This is done in PVS in the theory QueueRefine, see Figure 4.24. The variable declaration for c on line 4 uses the dependent types of PVS. Recall that ListQueueAssert? is a predicate on ListQueue coalgebras. By putting parenthesis around such a predicate one obtains the (sub–) type of those inhabitants that fulfil the predicate. So c is a ListQueue coalgebra (on state space X) that fulfils the method assertions of ListQueue. Technically, the declaration of c (together with the type parameters X and A) amounts to the assumption of an arbitrary ListQueue model.

The importing statement for QueueBasic makes all necessary notions from the specification Queue available. The functions abs_c and abs_new form the translation map ϕ (of Item 2 of Definition 4.10.1). The definition of abs_new looks a bit complicated because the interpretation z of the constructor of ListQueue cannot be declared as a variable.

It remains to prove the proposition model. The proof is not completely trivial because it involves some reasoning about bisimilarities. In the proof I used the following three utility lemmas:

```
bisim_char : Lemma Forall( x, y : X ) :
    bisim?(c)(x,y)   IFF   contents(c)(x) = contents(c)(y)

abs_bisim : Lemma Forall( x, y : X ) :
    bisim?(c)(x,y)   Implies   bisim?(abs_c(c))(x,y)

bisim_abs : Lemma Forall( x, y : X ) :
    bisim?(abs_c(c))(x,y)   Implies   contents(c)(x) = contents(c)(y)
```

The first one, bisim_char, gives a characterisation of bisimilarity on models of ListQueue: Two states are bisimilar precisely if their contents field is equal. The second lemma abs_bisim states that two bisimilar states x and y are also bisimilar when considered as states of a queue with respect to the translated coalgebra abs_c(c). The third lemma describes the converse situation.

Behavioural refinements are more difficult to construct in general because usually the existential quantifier, which is hidden in the notion of behavioural models, requires the construction of a suitable abstract model. However, the technical aspects of the translation of a behavioural refinement into PVS are as simple as for assertional refinements. For a behavioural refinement one would prove a proposition similar to the following.

```
same_behaviour : Proposition Forall( z : (ListQueueCreate?(c)) ) :
    bisim?(d(c), abs_c(c))(new(b(c)(z)), new(abs_new(c)(z)))
```

Here abs_c and abs_new form the translation map as before. The functions d and b give the abstract model, which is required in the notion of behavioural models. Usually the abstract model must be chosen in dependence of the concrete model, therefore d and b take the concrete model as arguments.

4.10.2. CCSL Case Studies

The following case studies have been performed with CCSL.

The MSMIE Protocol

The Multiprocessor Shared-Memory Information Exchange protocol (Bruns and Anderson, 1994) is a protocol for communication between several processors in a real-time control system. The protocol has been used for instance in the embedded software of Westinghouse nuclear system design. In (Meyer, 1999) an early version of CCSL is used to analyse the protocol. Meyer develops 4 specifications of the protocol in CCSL and proves several refinements. Finally he implemented the protocol in Java and uses an early version of the LOOP tool (Jacobs et al., 1998a) to translate the Java sources into (their semantics in) PVS. Then he proved that the Java program forms a valid model of the CCSL specification of the MSMIE protocol.

The YAPI Case Study

The Y–chart Application Programmers Interface (de Kock and Essink, 1999) is used at Phillips for the development of signal processing systems. For one aspect of the interface, the buffered data transfer, (Lambooij, 2000) develops a CCSL specification and shows its correctness via refinement.

Case Study on Transaction Mechanisms

Transactions are used to make certain designated sequences of actions atomic. Transaction mechanisms are important in the context of databases or operating systems, but also in the world of smart cards — where there is always the possibility that a smart card is removed before an appropriate sequence of actions is completed, see Chapter 5 of (Chen, 2000). In the joint work (Jacobs and Tews, 2001) we provide an abstract specification of a (simplified) transaction mechanism. There are two standard implementation techniques for a transaction mechanism: new value logging and old value logging. For both approaches we derive an assertional refinement from the original specification and prove its correctness in PVS. This case study uses the current CCSL compiler to translate the three specifications into PVS. The complete CCSL and PVS sources are available at the following URL: http://wwwtcs.inf.tu–dresden.de/∼tews/Transaction.

The Fiasco Case Study

Fiasco (Hohmuth, 2000; Hohmuth, 1998) is a micro kernel operating system developed within the DROPS (Härtig et al., 1998) project. The DROPS project is hosted at the computer science department of the Technische Universität Dresden (Dresden University of Technology) and aims on the construction of an operating system that supports quality of service requirements. As a micro kernel Fiasco implements only the absolutely necessary operating system functionality: address spaces, processes, and interprocess communication. Fiasco is an implementation of the L4 micro kernel interface in C++. It contains about 20.000 lines of C++ code.

In the Fiasco case study I formalised a part of the internal interface of the memory management in Fiasco. Then I tried to prove that the source code of Fiasco gives rise to a model of my specification. The case study revealed some hidden assumptions in the scrutinised interface, therefore the proof could not be completed. The case study is described in full detail in (Tews, 2000a), the source code (comprising all CCSL and PVS source files and also some C++ files) is available in the world wide web at http://wwwtcs.inf.tu-dresden.de/∼tews/vfiasco/.

For the Fiasco case study address spaces and virtual memory are important. Virtual memory is the memory that is visible to applications. Physical memory is the main memory that sits on the mother board of the computer. The operating system takes care that each application can use a fair amount of physical memory and that one application cannot access or modify the memory of another application without proper authorisation. This task is accomplished with *address spaces*. An address space defines a partial mapping of (addresses in) virtual memory to (addresses in) physical memory. Each application has its own address space so that the same virtual addresses usually refer to different physical addresses in different applications. Address spaces are *partial* mappings, because not all virtual addresses are mapped to physical addresses. If an application accesses such a non-mapped virtual address then the hardware signals a page fault. Page faults occur as the result of programming errors or when the operating system has swapped parts of the applications memory to hard disc. In the former case the application is usually terminated. In the latter case the operating system loads the data from the swap area and adjusts the address space of the application. If an application needs more memory it has to request it from the operating system. In case the request can be satisfied, the operating system changes the address space of the application. This shows that the manipulation of address spaces is a primary task for an operating system.

In an Intel based personal computer (more precisely in the IA32 architecture) the data structure that represents an address space is called a *page directory.* A page directory is a hierarchical structure of pointers that describe the address mapping. Any manipulation of address spaces boils down to the insertion or deletion of some pointers in a page directory. (For a more detailed account of virtual memory and address mapping see (Intel, 1999) or Chapter 4 of (Tews, 2000a).)

Fiasco is a particular nice challenge for ccsl because Fiasco was developed following the object–oriented paradigm: The whole micro kernel consists of a set of classes, each capturing some particular functionality. The class space_t provides the internal abstraction of address spaces: Objects of space_t are page directories and the methods of space_t provide suitable services. For instance the methods v_insert and v_delete insert or delete mappings of virtual memory (in the address space that is represented by the object on which these methods are invoked). Another method is switchin_context. It implements the change of the address space (by advising the hardware to use from now on the page directory in the current object for translating virtual addresses into physical ones).

In the case study I decided to investigate two correctness properties of the methods v_insert and switchin_context. The first correctness property is that these two methods should always terminate without itself producing a page fault. The second property is that after the insertion of a super page mapping[31] with a subsequent call of switchin_context the hardware should map the virtual addresses as desired.

To be able to express the two correctness properties I first developed the ccsl specification PhyMem of physical memory. The physical memory provides operations for reading and writing of memory cells. Further it is bounded, that is, accesses to addresses above a memory depended limit go into nowhere.

By exploiting inheritance of ccsl specifications, the physical memory is extended into a specification VirtMem of virtual memory. The virtual memory has a method virt_to_phy to translate virtual addresses into physical ones. The read and write methods are redefined as partial methods that work on virtual addresses now. They are partial methods because they fail if their virtual address argument is not mapped to a physical address by the address space in charge.

[31] A super page is a continuous memory area of 4 megabyte aligned at an address that is a multiple of 2^{22}.

The two specifications of virtual and physical memory form the basis of the case study. Therefore I checked their consistency and constructed the final model for both. The inspection of the state space of both models shows that there is no unwanted behaviour in the final models. This provides an informal argument for the correctness of the two memory specifications.

The specification Space_t captures a part of the interface of the class space_t. As assertions it contains the two correctness properties from above, which can now be expressed in terms of read and write operations on the virtual memory of VirtMem. The aim of the case study was then to show that the C++ source code of the methods v_insert and switchin_context of class space_t yields a model of the Space_t specification. For that I translated the C++ source by hand into PVS and tackled the proof. This sounds rather easy but the proof development in PVS was a three–month enterprise. The *hand translation* of the C++ sources into PVS is certainly a weakness of the case study. However, a translation tool that computes the semantics of a C++ program in the logic of PVS was certainly beyond the scope of the case study.[32]

As I indicated above the attempted proof failed because the space_t interface contains some hidden assumptions that have not been formalised in the specification Space_t. During the proof it became apparent that, if one combines certain states of the virtual memory with certain arguments of the method v_insert, then an assert statement[33] in the method v_insert fails (for a more precise formulation see Proposition 6.1 in (Tews, 2000a)). The hidden assumption in the interface of v_insert is that one may only insert address mappings for virtual addresses that are not already mapped. The source of FIASCO tests always for this condition.

The case study required about four months of work. It was carried out in the autumn of 1998 with PVS version 2.2, patch level 1.46. It consists of about 5,000 line of source code. There are 230 lemmas and 150 type correctness conditions that are proved with about 4,000 PVS commands. To type check the whole specification and to run all the proofs takes more than half a hour on a 333MHz Pentium II box.

Although the proof of the main theorem failed it was possible to verify a few properties of the FIASCO source code. Therefore it is fair to say that the case study was successful in the sense it showed that coalgebraic specification can well be applied to operating-system verification.

One has to say that PVS did not perform very well under this case study. During the case study I submitted numerous bug reports. Some time after I completed the case study PVS version 2.3 was released. This new version crashed when parsing the source code of the case study. At the time of writing PVS version 2.4 patch level 1 is available. It still contains some bugs that make it impossible to port the case study.

[32] Such a translation tool would of course be nonsensical unless it restricts C++ to a well understood subset.

[33] In C++ the assert statement (which is actually a preprocessor macro) tests for a logical condition and aborts the program if the condition is not true.

4.10.3. Specifying Companies — A Comparison with the UML

In this subsection I try to clarify the relation between CCSL and the *universal modelling language* UML with its *object constraint language* OCL. The UML is a mostly graphical software modelling language. An introduction to the UML is (Fowler, 1999), the current reference is (OMG, 2001). The object constraint language OCL is part of the UML, see (Warmer and Kleppe, 1999) or for the current reference (OMG, 1997). OCL is a formal language for specifying invariants, pre– and postconditions and other kinds of constraints.

The UML and its part OCL have a target different from CCSL: The UML is a tool dedicated to the software design process. As such UML necessarily contains informal elements, for instance text in natural language (which is not treated as a comment). However, any software design can also be viewed as a specification. This is the field where CCSL meets the UML.

At the time of writing a comparison between the UML and CCSL is impossible for the following reason: The UML is currently a moving target (Kobryn, 1999), whose design and definition process has not been completed yet. In particular, an official semantics of the UML does not exit (yet). The sections on semantics in the current reference manual (OMG, 2001) contain only additional syntactic constraints and explanations in natural language (Astesiano et al., 1999). It appears that currently for some UML constructs a different (and incompatible) semantics is used in different application domains of the UML.[34]

However, the UML has a large research community. There is work on the semantics of the UML (for instance (Clark et al., 2001)). Further, there is also work on topics that *require* a precise semantics for parts of the UML (for instance (Richters and Gogolla, 1998)). So instead of comparing CCSL with the official UML one could use one of the semantics of the UML that are currently in use. Unfortunately such a comparison is beyond the scope of this thesis.

In this subsection I illustrate the relation between CCSL and the UML/OCL by means of translating an UML example to CCSL. The translation is based on the semantics of UML class diagrams and OCL as it is described in (Richters and Gogolla, 1998) and (Warmer et al., 2001).

A translation of CCSL into (the formal part of) the UML is impossible for at least two reasons. First, the expressive power of OCL is far too low compared with CCSL's logic. Quantification in OCL is finite, because states of models of UML class diagrams are finite. So OCL is a *propositional language*: One can only express decidable properties in it (see also (Mandel and Cengarle, 1999) for the expressive power of OCL). Second, the UML is very much focused on the object-oriented paradigm. Similar to many object-oriented programming languages it lacks support for (algebraic) abstract data types.

For the example translation from the UML to CCSL I chose the company specification

[34] Jean–Michel Bruel at the UML tutorial of the ETAPS 2001 conferences in Genova.

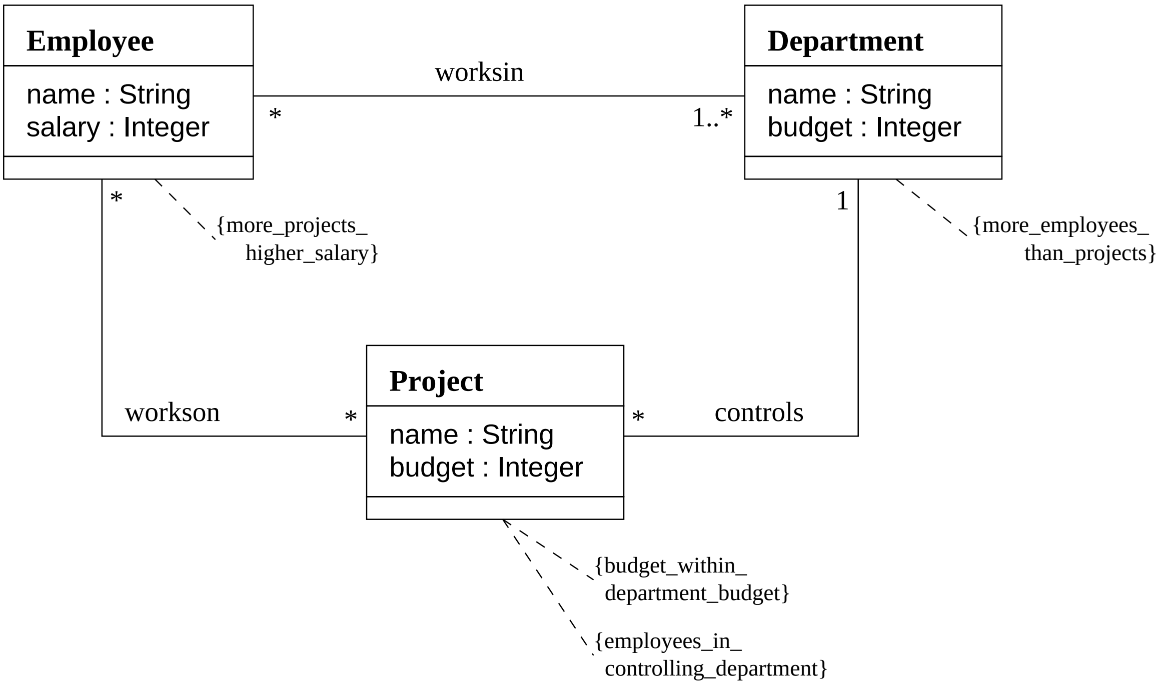
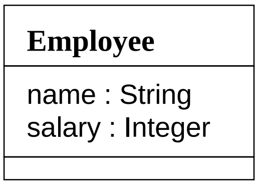


Figure 4.25.: A simple company — the original UML class diagram

that appears in the online documentation of the USE tool.[35] Almost the same example is also used in (Richters and Gogolla, 2002). Figure 4.25 shows the UML class diagram. For the translation to CCSL I will modify the example in the following. I refer to the version that is in Figure 4.25 (including the OCL constraints discussed below) as the *original company specification.*

Let me explain Figure 4.25 for those readers not familiar with the UML: Classes are displayed as boxes, like



Here, **Employee** is the name of the class and name and salary are its attributes. (The third frame in the box is reserved for method declarations. There are no methods in this example.) The lines between the classes are *associations*. An association is a relationship

[35] USE stands for UML-based Specification Environment, see http://www.db.informatik.uni-bremen.de/projects/USE/. The USE tool is also described in (Richters and Gogolla, 2002).

between instances of the two classes. Nothing is said about the kind of the relationship. An association can be as close as one object containing the associated object. However, it is also possible that an association merely indicates that the implementor of one class must be aware of the existence of the associated class.

The associations in Figure 4.25 are bidirectional, that is, from an instance of **Project** it is possible to get its controlling department *and* for an instance of **Department** one can get all controlled projects.

The annotation at the endpoints of associations are *multiplicities*. The multiplicity specifies how many instances of a class can participate in the association. More precisely, the multiplicity of an association end is the number of possible instances associated with a single instance of the other end. The multiplicity $*$ denotes an arbitrary (but finite) number.

Translated into English the associations in Figure 4.25 express that

- every employee works in at least one department (but there might be departments without employees)

- for every project there is precisely one controlling department (but there might be departments that control no project)

- employees can work on several projects (zero included) and several employees (also zero included) can work on one project

A model of an UML class diagram is an abstract state machine[36] ((Börger, 2002), Chapter 2 of (Stärk et al., 2001)). It contains (without going into details) a finite set of objects for each class in the diagram and a suitable relation for each association. Further, it supports all declared attributes and methods. Methods (which are not present in the current example) can change the current state, which might involve creating or deleting objects and changing attribute values.

In the original company example there are four OCL constraints that further restrict the class of models. Each constraint is associated with one class, its *context*. In Figure 4.25 I depicted the constraints with their names in braces. Expressed in English the constraints denote the following.

more_projects_higher_salary
Employees get a higher salary when they work on more projects.

more_employees_than_projects
The number of employees working in a department must be greater or equal to the number of projects controlled by the department.

[36] This is not explicit in (Richters and Gogolla, 1998) or (Warmer et al., 2001), there the authors spell out the details of a particular abstract state machine.

```
context Department
  inv more_employees_than_projects:
    self.employee–>size >= self.project–>size

context Employee
  inv more_projects_higher_salary:
    Employee.allInstances–>forAll(e1, e2 |
      e1.project–>size > e2.project–>size  Implies  e1.salary > e2.salary)

context Project
  inv budget_within_department_budget:
    self.budget <= self.department.budget

  inv employees_in_controlling_department:
    self.department.employee–>includesAll(self.employee)
```

Figure 4.26.: Original OCL constraints for the company example

budget_within_department_budget
The budget of a project must not exceed the budget of the controlling department.

employees_in_controlling_department
Employees working on a project must also work in the controlling department of that project.

Figure 4.26 shows the four constraints in OCL syntax. In OCL many things are implicit, so sometimes it is a bit difficult to understand what is meant. The context of an OCL formula determines the type of the free variable self. So in the first invariant more_employees_than_projects it denotes an object of class Department. The invariant holds in a state of a model, if it holds for all objects of class Department that are present in this state.

Operations for classes are written with the object-oriented dot notation: In the expression self.employee the operation employee is applied to self. The operation employee is implicitly defined for each Department object d: it yields the set of Employee objects that are associated via the association worksin with the department d in the current state. The operations project, department, and employee are implicitly defined in a similar way. There is an exception for associations endpoints with multiplicity 1: The operation that is implicitly defined for such associations returns an object instead of a set. Therefore, in the constraint budget_within_department_budget, the expression self.department denotes the controlling department (rather than a singleton set containing it).

For set-valued types (more precisely for all collections, but in this example the only collections are sets) the arrow –> replaces the dot in method invocation. So x–>size denotes the operation size (which is part of an assumed generic class for sets from the OCL standard library) applied to x. The result is the cardinality of x. The OCL constraints in Figure 4.26 use two other operations for sets: The operation forAll stands for universal quantification (over the variables e1 and e2) and includesAll denotes the subset relationship. Finely, the operation allInstances (which takes a class as argument) yields the set of all objects of that class that are present in the current state.

With these explanations it is easy to see that the constraints in Figure 4.26 coincide the English description given before.

In the following I formulate the company example in CCSL. There are two major problems: First, in CCSL the dependencies between classes must not be cyclic. Second, CCSL cannot represent bidirectional associations directly. Once these two problems are solved the translation into CCSL is straightforward. In the following I show two different translations of the company example into CCSL. In the first one I restructure the original UML specification avoiding circular dependencies. The second translation is more systematic and models the associations in a way such that circular dependencies are not problematic. I refer to the first translation as the *structured company specification* (because it contains some structure directly connecting the classes of departments and employees). The second translation is called the *flat company specification* (because the associations between departments, employees, and projects are moved to a different class).

For both the structured and the flat company specification, it is necessary to enrich CCSL with a type constructor for sets and a few operations over sets. Figure 4.27 shows the ground signature extensions that accomplish this. The first extension BaseTypes declares the types of strings, natural numbers, and ordinal numbers. It further declares a few comparison operations. CCSL does not support overloading, therefore it is a good idea to use different symbols for comparing natural numbers and ordinals. The types and constants of the BaseTypes ground signature are defined in the PVS prelude.

The second extension SetSig defines setof as a type constructor for sets. In what follows, both bisimulations and coalgebra morphisms do not play any role. Therefore, for this example, it is irrelevant whether I use the co– or the contravariant powerset functor for the semantics of setof (see also Remark 3.3.5). Here I use the contravariant version because it can be defined inside CCSL as a type equation.

The ground signature SetSig also declares some functions that are needed in the following. The function size determines the cardinality of a set. I defined size in an additional PVS theory because there is no such function available in PVS. It returns the number of elements for all finite sets and ω (the first limit ordinal) for all infinite sets. In the PVS prelude the function the is defined for singleton sets only. Here I declare it with a slightly more general type and rely on the PVS type checker (compare the discussion in Example 4.3.2).

The ground signature RelSig defines relations as subset of the cartesian product. The function domain returns the set of those x'es that stand in relation with a particular y.

```
Begin BaseTypes : GroundSignature
  Importing ordinal_defs

  Type string
  Type nat
  Constant              (* comparisons from the PVS prelude *)
    (>) : [nat,nat -> Bool];
    (<=) : [nat,nat -> Bool];

  Type ordinal
  Constant              (* comparisons for ordinals *)
    (>>) : [ordinal, ordinal -> Bool];
    (>>=) : [ordinal, ordinal -> Bool];
End BaseTypes

Begin SetSig[ X : Type ] : GroundSignature
  Importing Setops[X]

  Type setof = [ X -> Bool ]                   (* the type of sets *)

  Constant
    is_finite : [setof[X] -> Bool];            (* true for finite sets *)
    size : [setof[X] -> ordinal];              (* cardinality *)
    the : [setof[X] -> X];                     (* the({x}) = x *)
    subset? : [setof[X], setof[X] -> Bool];    (* subset relation *)
End SetSig

Type Rel[ X, Y : Type ] = [ X, Y -> Bool ]

Begin RelSig[ X, Y : Type ] : GroundSignature

  Constant              (* domain and codomain of relations as sets *)
    domain : [ Rel[X, Y], Y -> setof[X] ]
    domain(R,y) = Lambda(x : X) : R(x, y);

    codomain : [ Rel[X, Y], X -> setof[Y] ]
    codomain(R,x) = Lambda(y : Y) : R(x, y);
End RelSig
```

Figure 4.27.: CCSL ground signature extension with sets and ordinals

More formally:

$$\begin{array}{lclcl} \mathsf{domain}(R, y) & = & \coprod_{\pi_1} (R \wedge \pi_2^* \{y\}) & = & \{x \mid x\,R\,y\} \\ \mathsf{codomain}(R, x) & = & \coprod_{\pi_2} (R \wedge \pi_1^* \{x\}) & = & \{y \mid x\,R\,y\} \end{array}$$

In the first translation of the company example I restructure the original class diagram, thereby avoiding circular dependencies. Figure 4.28 shows an UML class diagram with the new class structure. There are the following differences with the original class diagram in Figure 4.25:

- Associations are directed now and have $*$ as multiplicity. This makes it possible to model each association with a set–valued method. To obtain the effect of the original multiplicities I add a few assertions in the resulting CCSL specification.
- The two OCL constraints in context project have been moved. This is necessary because, for instance, from an object of class Project one can not reach its controlling department any more.
- There is an additional class Configuration. This class captures the following difference in the semantics of CCSL and UML: A model of a class in CCSL is a set (of objects) together with a coalgebra. For CCSL the set of objects represents *all* possible (states of) objects. A model of an UML class diagram contains (among other things) a set of objects for each class in the diagram. However, for UML such a set of objects represents only the live objects in a certain state of the whole system.

  Objects of the class Configuration contain a set of departments, employees, and projects. Each Configuration object corresponds to a valid state in a model of the original class diagram.

The translation from Figure 4.28 to CCSL is rather straightforward: Each class in the diagram is modelled by one class specification in CCSL. Attributes are modelled by methods of the appropriate type. Associations are modelled by set–valued methods. Figure 4.29 shows the class interfaces in CCSL syntax.

It remains to translate the OCL constraints to CCSL and to add a few assertions that capture the multiplicities in the original company example. The assertions that deal with multiplicities are added to the class Configuration.

The OCL constraint more_projects_higher_salary can be translated without problems to CCSL. As an assertion in class Employee it looks as follows:

```
Assertion Selfvar x : Self
  more_projects_higher_salary :
    Forall(y : Self) : size(x.workson) >> size(y.workson) Implies
      x.salary > y.salary;
```

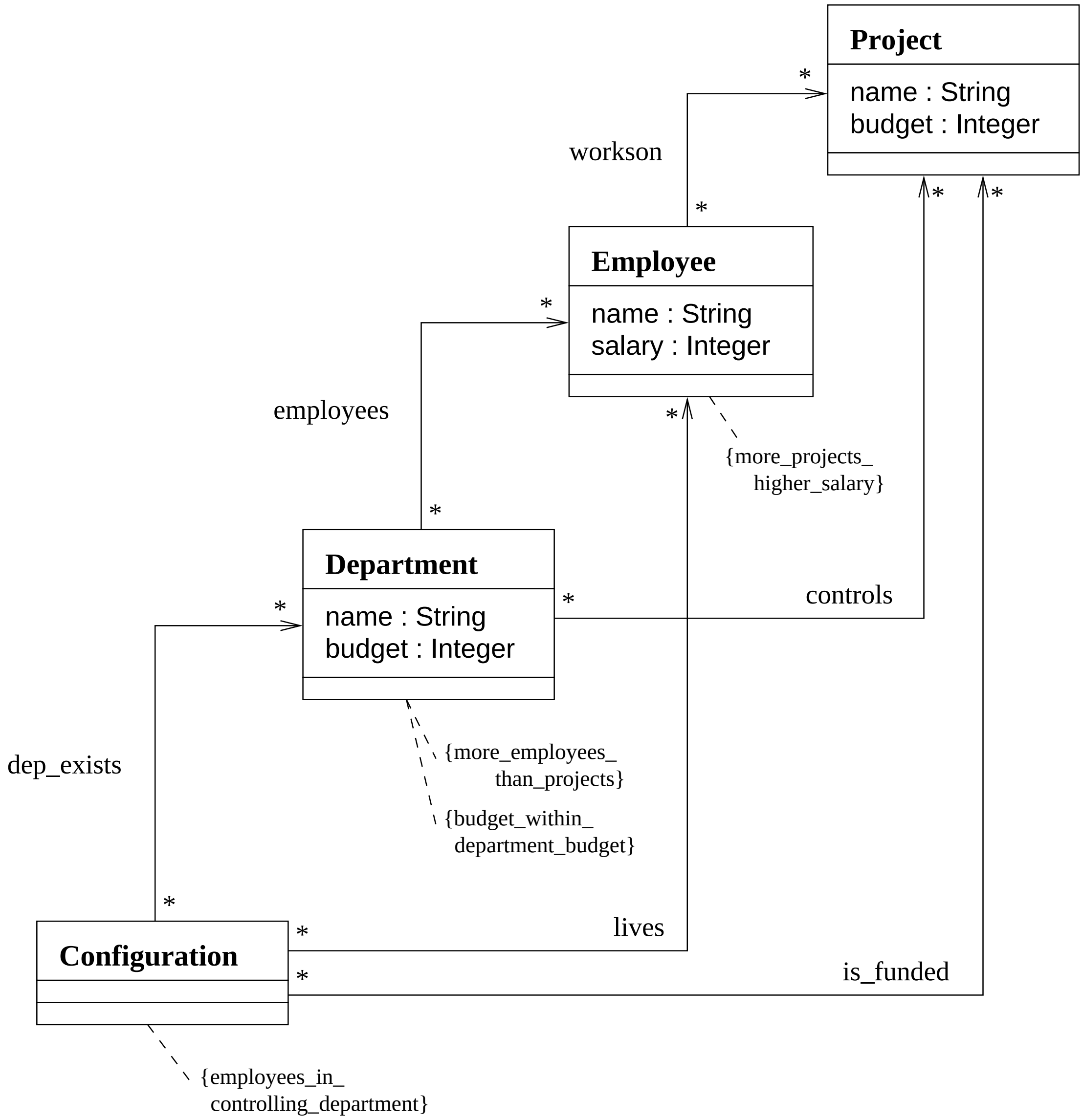


Figure 4.28.: A simple company — structured CCSL design

```
Begin Project : Final ClassSpec
  Method
    name : Self -> string;
    budget : Self -> nat;
End Project

Begin Employee : ClassSpec
  Method
    name : Self -> string;
    salary : Self -> nat;
    workson : Self -> setof[Project];
End Employee

Begin Department : ClassSpec
  Method
    name : Self -> string;
    budget : Self -> nat;
    controls : Self -> setof[Project];
    employees : Self -> setof[Employee];
End Department

Begin Configuration : ClassSpec
  Method
    is_funded : Self -> setof[Project];
    lives       : Self -> setof[Employee];
    dep_exists : Self -> setof[Department];

  Defining              (* return the employees that work on a given project *)
    project_members : [Self, Project] -> setof[Employee]
    project_members(s,p) = Lambda(e:Employee) : s.lives e And e.workson p;
End Configuration
```

Figure 4.29.: CCSL specification for the structured company (without assertions)

Note that the preceding assertion is not invariant with respect to behavioural equality, and indeed with this assertion the Employee specification does not have a final model.

The class department has two assertions:

```
Assertion Selfvar x : Self
  more_employees_than_projects :
    size(x.employees) >>= size(x.controls);

  budget_within_department_budget:
    Forall(p : Project) : x.controls p   Implies   p.budget <= x.budget;
```

Only the constraint budget_within_department_budget differs from the original OCL constraint, because it is situated in a different class now.

The OCL constraint employees_in_controlling_department has been moved to the class Configuration:

```
Assertion Selfvar s : Self
  employees_in_controlling_department :
    Forall(p : Project, d : Department) :
      s.dep_exists d And d.controls p Implies
        subset?( s.project_members p, d.employees);
```

The two preconditions account for the fact that the original OCL constraint is required only for those employees and departments that are present in a given state of the system.

The remaining assertions of the class Configuration deal with the multiplicities in the original class diagram or with facts that are defined in the semantics of UML class diagrams. First, the sets of objects that are present in a particular state must be finite:

```
finiteness :
  is_finite(s.is_funded) And is_finite(s.lives) And is_finite(s.dep_exists);
```

Next I capture the multiplicities. There are two requirements in the class diagram 4.25: First, each employee works in at least one department, and second, for every project there is a unique controlling department.

```
nonempty_affilation :
  Forall(e : Employee) : s.lives e Implies
    Exists(d : Department) : d.employees e And s.dep_exists d;

exists_controlling_department:
  Forall(p : Project) : s.is_funded p Implies
    Exists(d : Department) : s.dep_exists d And d.controls p;
```

```
unique_controlling_department :
  Forall(d1, d2 : Department) : Forall(p : Project) :
    s.dep_exists d1 And s.dep_exists d2 And s.is_funded p And
      d1.controls p And d2.controls p
    Implies
      d1 = d2;
```

The remaining three assertions deal with the fact that (the semantics of) an association (in a particular state) might only relate objects that exists (in that state).

```
workson_funded :
  Forall(e : Employee, p : Project) :
    s.lives(e) And e.workson p Implies s.is_funded p;

dep_control_funded :
  Forall(d : Department, p : Project) :
    s.dep_exists d And d.controls p Implies s.is_funded p;

dep_employees_live :
  Forall(d : Department, e : Employee) :
    s.dep_exists d And d.employees e Implies s.lives e;
```

This completes the structured company specification in CCSL. As part of this example I developed a model for the Configuration specification, thereby proving its consistency. The model and the necessary proofs are in the PVS source code for this section, see Appendix A.

The first translation into CCSL was obtained in a more or less *ad hoc* way. The resulting CCSL specification models some parts of the original UML diagram in a direct way: For instance the associations workson is represented by the method workson in the class Employee. This makes it possible to translate some of the original OCL constraints as assertions for the specifications of Department and Employee. For the structured company it is possible to develop, for instance, the theory of the class Employee (without taking departments into account). Note that this is not possible with an UML/OCL specification. The semantics of UML class diagrams and OCL constraints is only defined for complete systems. Reasoning about one class in isolation is impossible.

I present now the second translation of the company example into CCSL: The flat company specification. The structured company specification above resembles what one would have got when modelling the company from scratch in CCSL. In contrast, the flat company (developed below) resembles a specification *of the semantics* of the original UML class diagram.

Figure 4.30 shows the structure of the flat company as UML class diagram. This time associations are modelled as relations. They are stored in the class that models states of the complete system, which is called FlatCompany this time. As a consequence the OCL

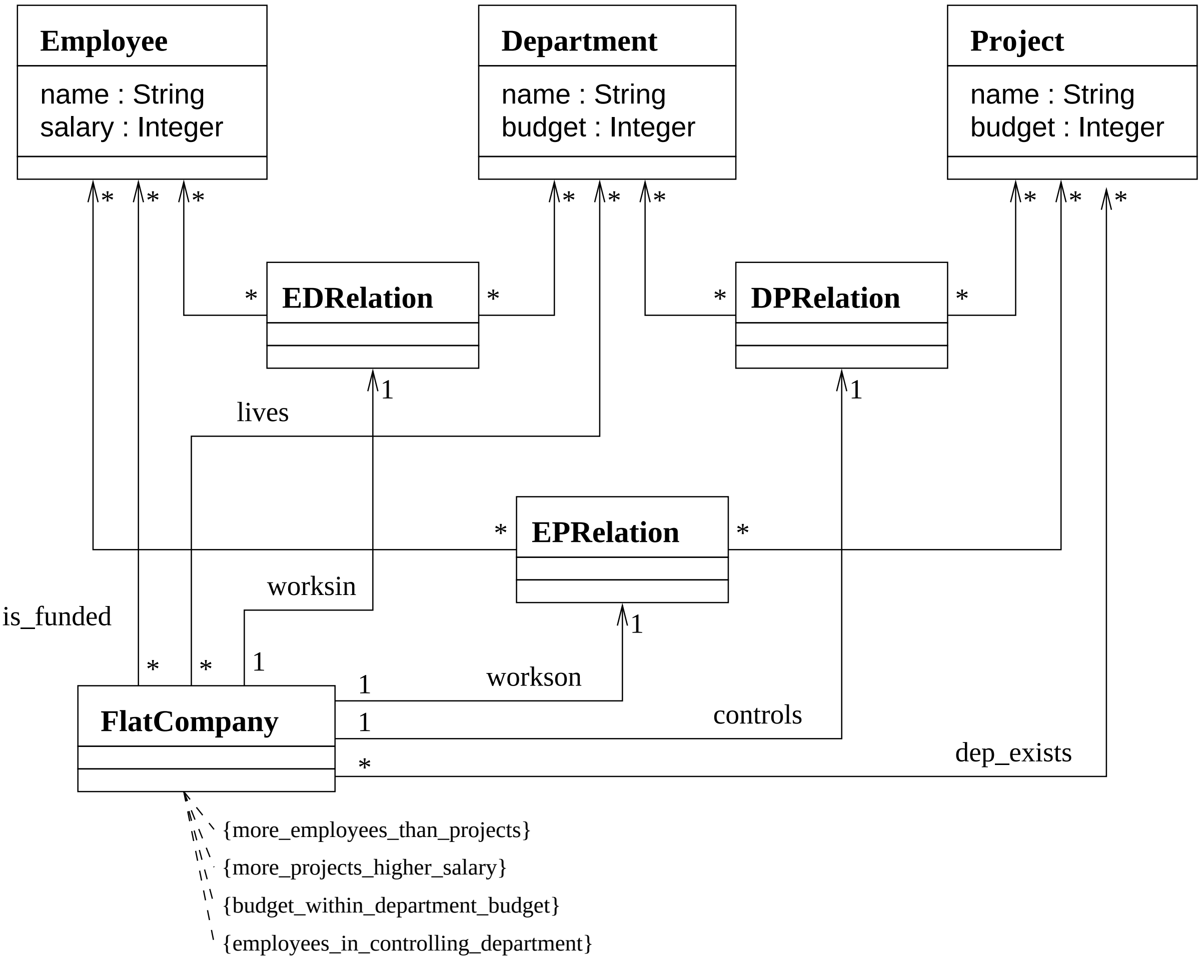


Figure 4.30.: A simple company — flat CCSL design

constraints must be moved to the class specification FlatCompany as well. The different relations that appear in diagram 4.30 are instances of the generic type of relations, which has been defined in Figure 4.27.[37] From diagram 4.30 the translation to CCSL is again straightforward. Figure 4.31 shows the result. In the CCSL source I prepended an F to the class names to make them distinct from the structured company specification. Note that it is now possible to use final semantics for projects, employees, and departments.

In the flat company only class FlatCompany contains assertions. The first assertion ensures that all system states are finite.

```
Assertion Selfvar s : Self
  finiteness :
    is_finite(s.is_funded) And is_finite(s.lives) And is_finite(s.dep_exists);
```

The next three assertions make sure that objects that are contained in one of the associations are also contained in the sets of 'living' objects.

```
worksin_living :
  Forall(e : FEmployee, d : FDepartment) : s.worksin(e,d) Implies
    s.lives e And s.dep_exists d;

controls_funded :
  Forall(d : FDepartment, p : FProject) : s.controls(d,p) Implies
    s.dep_exists d And s.is_funded p;

workson_living :
  Forall(e : FEmployee, p : FProject) : s.workson(e,p) Implies
    s.lives e And s.is_funded p;
```

The approach to represent an association with an relation does not model multiplicities. For the multiplicity constraints of the original company class diagram we need two additional assertions.

```
worksin_multiplicity :
  Forall(e : FEmployee) : s.lives e Implies
     Exists(d : FDepartment) : s.worksin(e, d);

controls_multiplicity :
  (Forall(p : FProject) : s.is_funded p Implies
       Exists(d : FDepartment) : s.controls(d,p))
  And
  (Forall(d1, d2 : FDepartment, p : FProject) :
    s.controls(d1,p) And s.controls(d2,p)
      Implies d1 = d2 );
```

[37] I prefer to use different names in Figure 4.30 for the three relations instead of introducing the UML construct for generic classes.

```
Begin FProject : Final ClassSpec
  Method
    name : Self -> string;
    budget : Self -> nat;
End FProject

Begin FEmployee : Final ClassSpec
  Method
    name : Self -> string;
    salary : Self -> nat;
end FEmployee

Begin FDepartment : Final ClassSpec
  Method
    name : Self -> string;
    budget : Self -> nat;
end FDepartment

Begin FlatCompany : ClassSpec
  Method
    is_funded : Self -> setof[FProject];
    lives       : Self -> setof[FEmployee];
    dep_exists : Self -> setof[FDepartment];

    worksin : Self -> Rel[FEmployee, FDepartment];
    controls : Self -> Rel[FDepartment, FProject];
    workson : Self -> Rel[FEmployee, FProject];
End FlatCompany
```

Figure 4.31.: CCSL specification for the flat company (without assertions)

It remains to translate the OCL constraints from Figure 4.26. The only thing that requires translation here is the access along an association. For instance the OCL expression self.employee in the more_employees_than_projects constraint is translated to domain(s.worksin, d). Here d takes the role of self of the current department and s is an object of class FlatCompany, which represents the current state of the whole system. The assertions are as follows:

```
more_employees_than_projects :
  Forall(d : FDepartment) : s.dep_exists d Implies
    size(domain(s.worksin, d)) >>= size(codomain(s.controls, d));

more_projects_higher_salary :
  Forall(e1, e2 : FEmployee) . s.lives e1 And s.lives e2 And
    size(codomain(s.workson, e1)) >> size(codomain(s.workson, e2))
    Implies e1.salary > e2.salary;

budget_within_department_budget :
  Forall(p : FProject) : s.is_funded p Implies
    p.budget <= the (domain(s.controls, p)) . budget;

employees_in_controlling_department :
  Forall(p : FProject) :
    subset?(domain(s.workson, p),
            mapflatten(domain(s.controls, p),
               Lambda(d:FDepartment) : domain(s.worksin,d)));
```

The last assertion uses the function mapflatten. It is defined in a separate ground signature extension in Figure 4.32. An application mapflatten(x, f) corresponds to the OCL term x−>collect(f)−>asSet() with operations from the OCL standard library.

The assertion budget_within_department_budget uses the function the (from the ground signature SetSig, see Figure 4.27) to access the only element in a singleton set. After translating the complete flat company specification to PVS this use of the will produce a *type check condition* (TCC) when type checking. The proof obligation (that the argument of the is a singleton set) can be proved with the help of the controls_multiplicity assertion. However, this requires that the FlatCompany specification is translated with the –dependent–assertions switch (see Subsection 4.9.9).

This completes the flat company specification. As for the structured company, the PVS source code contains a model for FlatCompany proving its consistency.

The translation that led to the flat company specification is done in a systematic way. One can translate a large class of UML class diagrams and OCL constraints into CCSL this way. However, the flat company specification is not so well structured: All semantic constraints and a large part of the structure of the original company (including all associations) are contained in the class FlatCompany.

As it stands the structured company and the flat company specification are not equivalent. There are the following differences:

```
Begin MapFlatSig[R,T : Type] : GroundSignature
  Importing MapFlat[R,T]
  Constant
    mapflatten : [setof[R], [R -> setof[T]] -> setof[T]]
    mapflatten(Rs, f) = Lambda(t : T) : Exists(r : R) : Rs(r) And f(r)(t)
End MapFlatSig
```

Figure 4.32.: Ground signature extension for mapflatten

1. In the structured company the constraint more_projects_higher_salary holds globally, that is for all employee objects. In the flat company this constraint is imposed only on *living* employees.

2. In the flat company two employees are provably equal when they have identical name and salary. However, in the structured company it is possible that two different employee objects have identical name and salary (and only their their field workson differs). A similar statement holds for departments.

3. In the flat company I used final semantics for departments and employees. This makes it possible to create (via the unique embedding into the final model) objects of type FEmployee and FDepartment. In the structured company I use loose semantics. Because there are no constructors it is impossible to create objects of type Employee and Department.

The preceding list of differences is complete. This has been proved in PVS. More precisely I proved the equivalence of the structured company and the flat company specifications under the following assumptions:

1. The constraint more_projects_higher_salary holds globally (that is for all employees) in the flat company.

2. In the structured company two employees are equal if they have equal names and salary and two departments are equal if they have equal names and budgets.

3. The loose models for employees and departments in the structured company are *big enough*: there exist coalgebra morphisms into the models of Employee and Department that allow me to identify employees and departments with particular attributes.

For a logical description of these assumptions and the precise equivalence that has been proved I refer to the PVS source code (see Appendix A).

The example done in this subsection suggests that UML class diagrams enriched with OCL constraints can be embedded into CCSL rather easily. The example shows the following conceptual differences between the UML and CCSL:

- The UML advocates the description of complete systems. Thus, mutual recursive dependencies between classes are no problem.
- CCSL focuses on single classes. It is necessary to work around cyclic dependencies. The structured company specification and the flat company specifications use two different approaches to deal with that problem.
- In CCSL less things are implicit. For a translation from the UML one has to make some parts of the UML semantics explicit.

4.11. Summary

This chapter described the coalgebraic class specification language CCSL. The unique feature of CCSL is the combination of abstract data type specification with coalgebraic class specifications that enables the iteration of (algebraic) abstract data types and (coalgebraic) behavioural types. The semantics of CCSL is based on algebras *and* on coalgebras. Other distinctive features of CCSL are the higher-order equational logic used in the class specifications and the method-wise modal operators.

This chapter is the most comprehensive account on CCSL, superseding (Hensel et al., 1998) and (Rothe et al., 2001). It describes the complete syntax and semantics of CCSL.

Comparison with other Specification Environments

OBJ is a family of similar algebraic specification languages based on order sorted algebra. Members of the family are for instance OBJ3 (Goguen and Malcolm, 1996), CafeOBJ (Diaconescu and Futatsugi, 1998), and BOBJ. The latter two include support for hidden algebras and reasoning about hidden congruences, so it is possible to specify behavioural types in CafeOBJ and in BOBJ. All members of the OBJ family contain parametrised modules and provide module expressions for the manipulation of modules. OBJ inherits the restrictions of algebraic specification. Most notably, signatures can only contain operations of the form $S_1 \times \cdots \times S_n \longrightarrow S_0$, where all the S_i are primitive sorts. Structured input and output types, as they occur naturally in CCSL, are impossible in OBJ.

The common framework initiative (Mosses, 1997) aims at a common framework for algebraic specification and development. It integrates many of the diverging extensions of algebraic specification. The initiative includes the design of the Common Algebraic Specification Language, CASL[38] (Mossakowski, 2000). The logic of CASL is a first-order

[38] The CASL language of the common framework initiative has nothing in common with the Custom Attack Simulation Language (CASL) (Vigna et al., 2000; Secure Networks, 1998).

logic with equality, partial functions, subsorting, and sort generation constraints. There are various sublanguages of CASL, for instance for conditional equational logic or horn-clause logic. However, CASL does only contain algebraic signatures, so structured input and output types are not possible. At the moment CASL does not support behavioural types.

DisCo (Kellomäki, 1997) is a specification method for reactive systems, it is developed at Tampere University of Technology, Finland. DisCo is based on the temporal logic of actions (TLA) (Lamport, 1994). In the DisCo specification language one can specify object systems and their transitional behaviour. Objects have a state chart like hierarchical structure but may not contain methods. Methods are specified outside of the objects as actions. DisCo specifications can be animated or translated into PVS. In PVS one can do refinement proofs, but it appears that crucial parts of the refinement proof cannot be formalised in PVS, because the translation of DisCo into PVS is incomplete (Kellomäki, 1997). There is no notion of abstract data type in the DisCo specification language.

The Unified Modelling Language UML is a graphical notion mainly developed for software design. However, in combination with the Object Constraint Logic OCL one can use UML class diagrams to specify properties of a software system. Such specifications can be equivalently described in CCSL. In general, a CCSL specification cannot be formulated as an UML class diagram.

Future Work

There are many ideas about how to develop CCSL further. Here are the more important ones:

- Support algebraic specifications (that is abstract data types with axioms).
- Generate more proofs for standard results.
- Include support for the powerset type constructor.

For the semantics of CCSL the open questions of Chapter 3 are important, especially how iterated specifications behave if they contain binary methods. Also type parameters with negative and mixed variance need clarification. Type parameters are important when they get instantiated with the special type Self. If a type parameter gets only instantiated with constant types, then it behaves more or less like an (unknown) constant. So it seems that in Theorem 4.7.4 one could allow type parameters with negative or mixed variance under the proviso that they get only instantiated with constant types. However, it is not clear how to get the technicalities right.

CCSL as presented here has the following disadvantages. The logic of CCSL is different from the logic of the target theorem prover. This is a consequence of developing the CCSL compiler as a front end to existing theorem provers. A possible solution would be to build a theorem prover for CCSL or to integrate CCSL into an existing theorem prover. The latter is an interesting idea in conjunction with the generic theorem prover ISABELLE. In principle CCSL could be integrated into ISABELLE/HOL in the same way as the data type package of (Berghofer and Wenzel, 1999).

The second disadvantage of developing CCSL as a front end is that type checking CCSL specifications is a two stage process. First, the CCSL compiler reads and checks a specification, then one has to load the generated theories into the target theorem prover. Some typing errors in the specification might only become apparent after the output of the CCSL compiler has been type checked in the target theorem prover. For instance a wrong use of an accessor function is not reported by the CCSL compiler. It only yields an unprovable type correctness condition in PVS.

5. Conclusion

The present thesis describes research in coalgebraic specification and verification. Coalgebraic specification is based on coalgebras and coinduction. Coalgebraic techniques are especially suited for the specification and verification of (possibly infinite) processes and object-oriented programs. The coalgebraic paradigm makes such specifications easier to understand. The provided proof principle coinduction helps to develop proofs on a more abstract level.

Coalgebraic techniques live in harmony with traditional algebraic specification. Indeed, it is best to combine algebraic and coalgebraic methods in one specification environment. Typically, in such an environment data structures are specified as abstract data types, while the entities that manipulate the data are specified with coalgebraic means.

The present thesis describes two contributions in the field of coalgebraic specification: Chapter 3 generalises the traditional notion of coalgebra such that coalgebras can model methods with arbitrary types, especially binary methods, as they occur in object-oriented programs. Chapter 4 presents the Coalgebraic Class Specification Language CCSL.

The approach of Chapter 3 to generalise the notion of coalgebra is bases on bivariant functors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \longrightarrow \mathbb{C}$ and the idea of separating co– and contravariant occurrences of the special type Self. This approach outclasses other approaches that have been proposed in the past to deal with the problem of binary methods in coalgebraic specifications. In this thesis I restrict the investigation to (what I call) higher-order polynomial functors. Higher-order polynomial functors are a straightforward generalisation of polynomial functors, built up from the identity, constants, products, coproducts, and (unrestricted) exponents.

Besides the definition of the generalised notions of coalgebra, coalgebra morphism, bisimulation, and invariant Chapter 3 performs a careful analysis of the properties of the new notion of coalgebra (and thereby its usefulness). Some of the results have been expected (for instance the absence of the final coalgebra if binary methods are present), other results are rather surprising (for instance that invariants do not give rise to subcoalgebras).

The analysis in Chapter 3 yields different levels of generalisation: Higher levels of generalisation can model more complicated method types, but in turn possess less properties. Coalgebras for extended polynomial functors seem to be a fair compromise: They provide sufficient generality to model classes of mainstream object-oriented languages like C++, Java, or Eiffel. Still, coalgebras for extended polynomial functors possess most of the structural properties that are known to hold for the traditional notion of coalgebra. If the existence of a greatest bisimulation is required, then one has to work with the more restricted class of extended cartesian functors.

An additional feature of the present thesis is that many theoretical results and all the examples have been *mechanically verified* with the theorem prover PVS. The source code of the PVS formalisation and the proofs are available in the world wide web.

Chapter 4 of the present thesis is a comprehensive account on the Coalgebraic Class Specification Language CCSL, it describes its syntax and semantics in detail. CCSL combines algebraic specification with coalgebraic specification. The design of CCSL and the implementation of a prototype compiler was a group effort in the LOOP project on formal methods for object orientation. CCSL uses initial algebras as semantics of abstract datatype specifications and coalgebras as semantics of class specifications. One can use final semantics (i.e., final coalgebras) for class specifications, if desired. The logic of CCSL is based on higher-order logic with two extensions: behavioural equality and method-wise infinitary modal operators. CCSL incorporates the extended notion of coalgebra from Chapter 3 of the present thesis as well as the results on iterated specifications from the related work of (Hensel, 1999) and (Rößiger, 2000b). CCSL is the only specification language, that I am aware of, that can be used with either one of the two leading theorem provers for higher-order logic PVS and ISABELLE/HOL. CCSL has been applied in several case studies, the most spectacular one on the verification of the micro-kernel operating system FIASCO.

CCSL and its prototype implementation are research tools in that they allow the user to experiment with signatures whose structural properties have not been investigated yet. Novice users can request the CCSL compiler to check that their specification stay in the well understood fragment of CCSL. As a tool coming out of a PhD project CCSL could be improved in many ways. In a future version CCSL should support axioms in algebraic specifications. Further the CCSL compiler should generate proofs along with the lemmas it generates.

Appendix

A. Guide to the PVS Sources

The theorem prover PVS plays in several ways an important role for this thesis. First, important parts of the main propositions of Chapter 3 have been formalised and proved in the higher-order logic of PVS. Second, PVS is one of the target systems of the prototype compiler for the coalgebraic class specification language CCSL. Third, a number of examples that are presented in this thesis have been developed with PVS. So there is a lot of PVS material that has been developed partly to prove results of the this thesis, partly to illustrate the presented material. This appendix gives some more information about the PVS material that is related to this thesis.

The PVS sources (and the CCSL sources, if applicable) are available in the world wide web at the following URL:

http://wwwtcs.inf.tu–dresden.de/∼tews/PhD/

The CCSL compiler is available at

http://wwwtcs.inf.tu–dresden.de/∼tews/ccsl/

and the theorem provers PVS and ISABELLE are at

http://pvs.csl.sri.com/ http://isabelle.in.tum.de/

The preceding web pages contain additional information about how to down load, install, and run the software. The main reference for PVS is (Owre et al., 1996), but as introduction I recommend the documentation shipped with the system (Owre et al., 1999a; Owre et al., 1999b) and the tutorials (e.g., (Crow et al., 1995)) available via the PVS home page at the preceding URL. ISABELLE/HOL is described in (Nipkow et al., 2002b). Additional technical descriptions are in the manuals (Nipkow et al., 2002a; Wenzel, 2002; Paulson, 2002).

The PVS sources comes in four directories. Their contents is as follows:

Fibration contains the formalisation of fibred category theory that is used in Section 2.4 and Chapter 3,

Queue contains the running example of queues of Chapter 3,

UML contains the CCSL translations of the company example of Subsection 4.10.3,

Variance contains a formalisation of the variance algebra of Subsection 4.2

```
Unit : Theory                          % the final object in the PVS universe
Begin
  unit : Type = upto(0)
End Unit

Bang[X : Type] : Theory                % unique morphism into the final
Begin
  Importing Unit
  bang : [X -> unit] = Lambda( x : X ) : 0
  single_fun : Lemma Forall( f : [X -> unit] ) : f = bang
End Bang
```

Figure A.1.: Example PVS source code from file `base.pvs`: The final object in PVS.

The second section of this appendix contains a table that relates examples, propositions, and lemmas of this thesis with the PVS source code. The following section illustrates the formalisation of fibred category theory that has been used to prove the main theorems of Chapter 3 in PVS.

A.1. Details of the PVS Formalisation

In the following I do not pay much attention to the concrete syntax of the PVS specification language, because it is straightforward to understand. For instance the lambda calculus expression $\lambda x : A \,.\, f(x)$ translates to **Lambda**$(x : A) : f(x)$ in the syntax of PVS. Similarly for existential and universal quantifiers. Syntactical peculiarities and bits of the semantic of PVS are explained along the way. In the following, PVS source code is set in sans serif, PVS keywords are in **sans serif bold extended**. Comments start with a percent sign % and reach till the end of the line, they are set in *italics*.

The PVS source files are organised in *theories*. Theories have a unique name, can declare parameters, and contain declarations and theorems. Declarations define types, constants and functions. Theorems are formulae (i.e., boolean expressions) that can be proved to be true in PVS. Figure A.1 shows as an example two theories that formalise the final object in PVS.

The theory Unit has no type parameters and contains only one declaration: Via the PVS built-in upto[1] the type unit is defined as the type of all natural numbers up to (and containing) zero, that is, unit is the type with the only inhabitant 0.

[1] The type upto is defined in the PVS prelude, which contains material that is available in all theories. The type upto is a dependent type, it maps any natural number i to the type containing all j with $j \leq i$ as inhabitants.

The theory Bang proves that unit is indeed a final object. It contains a type parameter that stands for an arbitrary other object X. The importing clause makes the material from theory Unit available in theory Bang. The declaration bang defines the function $!_{\mathsf{X}}$ (thus proving its existence) and the theorem single_fun states its uniqueness. The function definition bang must declare the type of bang, here $\mathsf{X} \Rightarrow \mathsf{unit}$, in PVS written as [X –> unit]. The proof of the theorem single_fun applies first the extensionality rule of higher-order logic:

$$\frac{\Gamma \vdash \forall x \,.\, f\,x = g\,x}{\Gamma \vdash f = g}$$

Then one expands the definition of bang and uses the type information to show that $f\,x$ equals zero.

Type parameters as in theory Bang add a form of polymorphism to PVS. Inside a theory a type parameter functions as a constant type. From outside it looks like a type variable: A theory that imports Bang can use the function bang with an arbitrary type instantiated for X. The typechecker of PVS tries to infer the instantiation of a polymorphic declaration. If this fails the user has to provide it explicitly. PVS usually finds the right instantiation for declarations that depend on all type parameters of the theory in which they are defined. Therefore it is preferable that all declarations in one theory depend on all type parameters. For this reason it is often the case that strongly related material is split into several theories with a different number of type parameters.

The dependent types in PVS make type checking undecidable. Therefore the typechecker of PVS generates *type check conditions* (also called TCC's). A TCC is associated with a subexpression in the input that cannot be type checked to be correct. Type check conditions have the form of theorems and the user is required to prove them, thereby helping the typechecker out. For instance for the function definition bang in theory Bang PVS generates the following TCC:[2]

bang_TCC1 : **Obligation Forall**(x : X) : 0 <= 0

The reason here is that the typechecker cannot check if the zero in the definition of bang is an inhabitant of type unit. Therefore we have to prove that 0 fulfils the defining predicate of upto(0). Most TCC's (in particular the preceding one) can be discharged with the default PVS TCC proof strategies.

In higher-order logic one models predicates by their characteristic functions. PVS provides the following predefined type for predicates (where T is a type parameter):

PRED : **TYPE** = [T –> bool]

So a predicate $(\mathsf{P} \subseteq \mathsf{X})$ translates to a declaration P : PRED[X]. PVS directly supports comprehension through its predicate subtyping feature. For a predicate $\mathsf{P} \subseteq \mathsf{X}$ (i.e.,

[2] This TCC is generated by PVS version 2.3. Version 2.4 does not generate any TCC's for the two theories in Figure A.1, presumably because the type checker in the newer version is more sophisticated.

for a function P : X$\longrightarrow$bool) the type expression (P) denotes the type comprising all inhabitants of X that fulfil P (i.e., those that P maps to true).

The first step in the formalisation is the construction of the bicartesian closed structure in the base category $\mathbb{B}_{\mathrm{PVS}}$. (Recall from Section 2.4.4 that $\mathbb{B}_{\mathrm{PVS}}$ is formed by the types and functions that can be represented in PVS.) The corresponding PVS theories are in file `base.pvs`. I discuss coproducts in detail and sketch products and exponents below. The initial object is formalised by the empty type, the morphism out of the initial object is the empty function. Both do exist in PVS's version of higher-order logic. In the following I show how one can obtain coproducts.

The coproduct of two types can be formalised with the abstract datatype feature of PVS, because in **Set** the coproduct of X and Y can be represented as the carrier of the initial algebra for the (constant) functor $X \longmapsto A + B$.[3] In order to define an abstract data type in PVS one has to give its signature in a syntax similar to those of the theories in PVS. For the coproduct this has the following form:

```
Coproduct[X, Y : Type] : Datatype      % define Coproduct
Begin
  in1(acc1 : X) : in1?
  in2(acc2 : Y) : in2?
End Coproduct
```

When PVS processes this it defines a new parametric type Coproduct, a number of functions, and some axioms that ensure that the type Coproduct[X, Y] is the carrier of the initial algebra for the coproduct signature. In particular it defines two constructors

```
in1 : [ X -> Coproduct[X, Y]]
in2 : [ Y -> Coproduct[X, Y]]
```

that correspond to the injections κ_1 and κ_2. Additionally, there are two recogniser predicates

```
in1?, in2? : [Coproduct[X, Y] -> boolean]
```

(In PVS identifiers can contain question marks.) The recognisers deliver true for an inhabitant of Coproduct$[A, B]$ precisely if it was built via the respective constructor. The accessor functions acc1 and acc2 allow one to destruct an inhabitant of the coproduct type. Naturally, the accessor acc1 can only be applied to inhabitants of Coproduct$[A, B]$ that have been built with the in1 constructor, similarly for acc2. The accessors get a precise type using the predicate subtypes of PVS:

```
acc1: [( in1? ) -> X]
acc2: [( in2? ) -> Y]
```

[3] This seems circular but is not, because the functor can be represented without the coproduct as a signature with two constructors.

```
CoproductPair[X,Y,Z : Type] : Theory        % define copairing
Begin
  Importing Coproduct[X,Y]
  f : Var [X -> Z]
  g : Var [Y -> Z]

  copair(f,g) : [Coproduct[X,Y] -> Z] =
    Lambda(xy : Coproduct[X,Y]) : Cases xy OF
      in1(x) : f(x),
      in2(y) : g(y)
    EndCases

  copair_char1 : Lemma
    copair(f,g) o in1 = f    And   copair(f,g) o in2 = g

  copair_char2 : Lemma Forall( h : [Coproduct[X,Y] -> Z] ) :
        h o in1 = f And h o in2 = g      Implies    h = copair(f,g)
End CoproductPair
```

Figure A.2.: Definition and properties of copairing in PVS (file `base.pvs`)

As next I define the copairing of two morphisms $\mathsf{X} \xrightarrow{f} \mathsf{Z} \xleftarrow{g} \mathsf{Y}$. This construction is parametric in the three objects X, Y, and Z, so it is placed in a theory with three type parameters. The PVS source code is in Figure A.2.

The theory CoproductPair imports the datatype Coproduct with a concrete instantiation. The fourth and the fifth line in Figure A.2 declare two variables f and g (of functional type). In PVS variable declarations are syntactic sugar that can save a considerable amount of source code. Without the variable declarations the definition of copair would be

```
copair : [[[X -> Z], [Y -> Z]] -> [Coproduct[X,Y] -> Z]] =
  Lambda( f : [X -> Z], g : [Y -> Z] ) :
    Lambda( xy : Coproduct[X,Y] ) : Cases xy OF
                              ...
```

In the definition of copair I use the **Cases** ··· **OF** ··· **EndCases** expression that is provided to examine values of an abstract data type. Here, the value xy is matched against the two constructor expressions and the matching branch yields the result.

The lemma copair_char1 expresses that copair(f,g) makes Diagram 2.1 (from page 19) commute. Lemma copair_char2 proves that copair(f,g) is the unique function with this property. The variables f and g that occur free in both lemmas are implicitly universally quantified. The infix operator **o** denotes function composition in PVS.

The coproduct of two morphisms is a construction that is parametric in four objects, so it is placed in a separated theory with four type parameters U, V, X, and Y. With two variable declarations the coproduct of morphisms is defined as follows (from now on I skip the theory header and display only the relevant declarations and theorems):

```
f : Var [U -> X]
g : Var [V -> Y]
plus( f, g ) : [Coproduct[U,V] -> Coproduct[X,Y]] = copair( in1 o f, in2 o g )
```

In **Set** one can prove that the morphism $f+g : U+V \longrightarrow X+Y$ does a case distinction: If the argument comes from U then f is applied, otherwise g. In PVS the corresponding theorem is

```
plus_char : Lemma plus(f,g) =
  Lambda( uv : Coproduct[U,V] ) : Cases uv OF
    in1(u) : in1(f(u)),
    in2(v) : in2(g(v))
  EndCases
```

My definition of plus follows very closely the categorical definitions. However, the decision procedures of PVS are tuned for element-wise computations, so they would perform better if the property in the plus_char lemma would be the definition for plus. Still, I decided to follow in all definitions as closely as possible the original definitions in category theory. This makes it much easier to check that the definitions are correct. Most definitions are followed by an element-wise characterisation lemma like plus_char above. These characterisation lemmas can be used in proofs to improve the performance.

This completes the construction of finite coproducts in the category $\mathbb{B}_{\mathrm{PVS}}$. Let me now sketch products and exponents. The tuple type of PVS forms a product in $\mathbb{B}_{\mathrm{PVS}}$. For two types X and Y the tuple type is written with square brackets as [X, Y]. PVS provides projections as special expressions. To get the projections as functions I use the following theory:

```
Pi_12[ X, Y : Type] : Theory
Begin
  pi_1( x : X, y : Y ) : X = x
  pi_2( x : X, y : Y ) : Y = y
End Pi_12
```

The product of two functions is defined via pairing:

```
pair: [[[Z -> X], [Z -> Y]] -> [Z -> [X,Y]]] =
  Lambda( f : [Z -> X], g : [Z -> Y] ) : Lambda( z : Z ) : (f(z), g(z))
```

```
f : Var [U -> X]
g : Var [V -> Y]
times( f, g ) : [[U,V] -> [X,Y]] = pair(f o pi_1[U,V], g o pi_2[U,V])
```

Then one can prove

```
times_char : Lemma
  times( f, g ) = Lambda( uv : [U,V] ) : ( f(pi_1(uv)), g(pi_2(uv)) )
```

The built-in function type is the exponent in $\mathbb{B}_{\mathrm{PVS}}$. It is written as [X -> Y] for two types X and Y. The constructions on morphisms are as follows (compare the description of exponentials on page 19):

```
eval( f : [X -> Y], x : X ) : Y = f(x)

abstr( g : [[Z, X] -> Y] ) : [Z -> [X -> Y]] =
  Lambda( z : Z ) : Lambda( x : X ) : g(z, x)

f : Var [V -> U]
g : Var [X -> Y]
=>( f, g ) : [[U -> X] -> [V -> Y]] = abstr(g o eval o times( id[[U->X]], f) )

exp_char : Lemma
  (f => g) = Lambda(h : [U -> X]) : Lambda( v : V ) : g(h(f(v)))
```

The special syntax =>(f, g) defines => as infix operator. The function id[[U->X]] is the identity function on the function space [U->X].

This completes the cartesian closed structure of $\mathbb{B}_{\mathrm{PVS}}$. In addition the PVS sources contain a formalisation of pullbacks (which I skip here).

For the predicate fibration the fibre over an object $X \in |\mathbb{B}_{\mathrm{PVS}}|$ is represented by the type PRED[X]. For the fibration of relations one takes PRED[[X, Y]]. The bicartesian closed structure of an arbitrary fibre is defined in a theory with one type parameter X:

```
P,Q : Var PRED[X]

truth : PRED[X] = Lambda(x : X) : True;
falsehood : PRED[X] = Lambda(x : X) : False;

fib_and( P, Q ) : PRED[X] = Lambda(x : X) : P(x) And Q(x)
fib_or( P, Q ) : PRED[X] = Lambda(x : X) : P(x) OR Q(x)
fib_exp( P, Q ) : PRED[X] = Lambda(x : X) : P(x) Implies Q(x)
```

(This and the following material is from file `fibration.pvs`.) The single fibres are preorder categories (i.e., between any two predicates there is at most one morphism). The morphisms are provable entailments; they are captured as follows

```
>>( P,Q ) : bool = Forall(x : X) : P(x) Implies Q(x)
```

This defines the infix operator >> as relation on predicates such that P >> Q holds precisely if there is a morphism $\mathsf{P} \longrightarrow \mathsf{Q}$ in the fibre over X.

The definition of the substitution functor f^* and its adjoints $\coprod_f$ and $\prod_f$ requires an additional type parameter Y:

```
f :  Var [X -> Y]
star(f) : [PRED[Y] -> PRED[X]] = Lambda(Q : PRED[Y]) :
    Lambda(x : X) : Q(f(x))

coprod(f) : [PRED[X] -> PRED[Y]] = Lambda(P : PRED[X]) :
    Lambda(y : Y) : Exists(x : X) : P(x) And y = f(x)

prod(f) : [PRED[X] -> PRED[Y]] = Lambda(P : PRED[X]) :
    Lambda(y : Y) : Forall(x : X) : y = f(x) Implies P(x)
```

That coprod and prod are indeed adjoints of star is proved by

```
left_coprod : Lemma
  P >> star(f)(Q)    IFF    coprod(f)(P) >> Q

right_prod : Lemma
  Q >> prod(f)(P)    IFF    star(f)(Q) >> P
```

These examples show how constructions and properties in the total category are translated to PVS: Every fibre, which is used, yields one type parameter (the object in the base). Arbitrary objects from the fibres and morphisms from the base category are declared as variables.

The next construction is the bicartesian closed structure in the total category of predicates (again I use infix operators):

```
P : Var PRED[X]
Q : Var PRED[Y]

/\(P,Q) : PRED[[X,Y]] =
      fib_and( star(pi_1[X,Y])(P), star(pi_2[X,Y])(Q) )

\/(P,Q) : PRED[ Coproduct[X,Y] ] =
      fib_or( coprod(in1)(P), coprod(in2)(Q) )

=>(P,Q) : PRED[[X -> Y]] =
  prod(pi_1[[X->Y],X]) (fib_exp( star(pi_2[[X->Y],X])(P), star(eval)(Q)))
```

Note that here the infix operator => gets overloaded for predicates, very much the same as ⇒ is used in the present thesis.

To facilitate proofs there are the following characterisation lemmas:

```
pred_prod_char : Lemma (P /\ Q) =
  Lambda( x : X, y : Y) : P(x) And Q(y)


pred_coprod_char : Lemma (P \/ Q) =
  Lambda( x : Coproduct[X,Y] ) : Cases x OF
      in1(y1) : P(y1),
      in2(y2) : Q(y2)
  EndCases

pred_exp_char : Lemma (P => Q) =
  Lambda(f : [X -> Y]) : Forall(x : X) : P(x) Implies Q(f(x))
```

The bicartesian closed structure of relations is similarly easy to define, it only looks a bit more complicated. For instance the following declarations define the exponent of relations in a theory with four type parameters U, V, X, and Y:

```
S : Var PRED[[U,V]]
R : Var PRED[[X,Y]]

relexp( S, R ) : PRED[ [[U->X], [V->Y]] ] =
  prod( times(pi_1[[U->X], U], pi_1[[V->Y], V]) )
      (fib_exp( star(times(pi_2[[U->X], U], pi_2[[V->Y], V]))(S),
                star(times(eval[U,X],eval[V,Y]))(R)))

rel_exp_char : Lemma relexp( S, R ) = Lambda( f : [U->X], g : [V -> Y] ) :
  Forall(u : U, v : V) : S(u,v)   Implies   R(f(u), g(v))
```

With these definitions one can express already many Lemmas from Subsection 2.4. For instance, in PVS Lemma 2.4.7 from page 43 (stating that truth preserves the bicartesian structure) has the following form (see theory `PredProps` in file `fibprops.pvs`):

```
truth_prod : Lemma (truth[X] /\ truth[Y]) = truth[[X,Y]]


truth_coprod : Lemma (truth[X] \/ truth[Y]) = truth[Coproduct[X,Y]]

truth_exp : Lemma (truth[X] => truth[Y]) = truth[[X -> Y]]
```

All three statements are proved by grind after applying extensionality. The more interesting Lemma 2.4.17 (about cofibredness on page 47) is translated to PVS as follows (see theory `FibProps`):

```
f : Var [X -> U]       %   Note:
g : Var [Y -> V]       %   (f × g)    : (X × Y)   -> (U × V)
P : Var PRED[X]        %   (f + g)    : (X + Y)   -> (U + V)
Q : Var PRED[Y]        %   (V => f) : (V => X) -> (V => U)

cofibred_product : Lemma
  coprod( times(f,g) )( P /\ Q )   =   ( coprod(f)(P) /\ coprod(g)(Q) )

cofibred_coproduct : Lemma
  coprod( plus(f,g) )( P \/ Q )   =   ( coprod(f)(P) \/ coprod(g)(Q) )

const_exp : Lemma
   coprod( id[V] => f )( truth[V] => P )   =   ( truth[V] => (coprod(f)(P)) )
```

The Lemmas 2.4.9 and 2.4.10 (on page 43f) state facts about intersection and union over arbitrary collections of predicates. Assume that the predicates are predicates over the type (or the object) Base and that the index set is given as a type Index. Then a collection of predicates ($P_i \subseteq \mathsf{Base})_{i \in \mathsf{Index}}$ can be formalised as a function $\mathsf{Index} \longrightarrow \mathsf{PRED[Base]}$. Intersection and union over such collections can be expressed with universal and existential quantification, respectively. This is done in theory `Collection` in file `base.pvs`. The PVS source code is as follows (Base and Index are type parameters):

```
collection : Type = [Index -> PRED[Base]]

Coll_and( c : collection ) : PRED[Base] =              % intersection
  Lambda( b : Base ) : Forall( i : Index ) : c(i)(b)

Coll_or( c : collection ) : PRED[Base] =               % union
  Lambda( b : Base ) : Exists( i : Index ) : c(i)(b)
```

With these definitions Lemma 2.4.9 can be formalised in a theory with three type parameters X, Y, and Index. The first equation of 2.4.9 (1) looks as follows (see again theory `PredProps`):

```
Pi : Var collection[X, Index]
Qi : Var collection[Y, Index]

coll_and_prod : Lemma
  Coll_and( Lambda( i : C ) : Pi(i) /\ Qi(i)) =
     ( Coll_and(Pi) /\ Coll_and(Qi) )
```

As next I show how the functor T from Example 3.3.8 (on page 88) is formalised in PVS. The PVS source is in file `t.pvs`. The object part is captured by a type definition that is parametric in Y and X:

```
TIface : Type = [[X -> Y] -> X]
```

This way we have $T(\mathsf{Y},\mathsf{X}) = \mathsf{TIface}[\mathsf{Y},\mathsf{X}]$. The morphism part is

```
f : Var [U -> X]
g : Var [Y -> V]

tf( g, f ) : [ TIface[V,U] -> TIface[Y,X] ] = ((f => g) => f)
```

(I omit the lemma t_fun_char that characterises tf as a λ–expression.) Now, the declarations

```
c : Var [X -> TIface[X,X]]
d : Var [Y -> TIface[Y,Y]]
```

stand for two T–coalgebras: c on carrier X and d on Y. With the exponents of predicates and relations it is straightforward to define T–invariants and bisimulations.[4]

```
P : Var PRED[X]
Q : Var PRED[Y]

PredT(Q,P) : PRED[TIface[Y,X]] = ((P => Q) => P)
T_invariant?(c)(P) : bool = Forall( x : X ) : P(x)   Implies   PredT(P,P)(c(x))

S : Var PRED[[U,V]]
R : Var PRED[[X,Y]]

RelT(S,R) : PRED[ [TIface[U,X], TIface[V,Y]] ] = relexp(relexp(R,S), R)
T_bisimulation?(c,d)(R) : bool = Forall(x : X, y : Y) :
   R( x, y )    Implies    RelT(R,R) ( c(x), d(y) )
```

The declaration T_invariant? is a recogniser for T–invariants. Applied to a coalgebra c and a predicate P it delivers true if and only if P is an invariant for c. Similarly for T_bisimulation?.

Figure A.3 shows the PVS sources for Example 3.3.8. The first two lines define A and B as enumeration types with four elements.[5] The rest of theory InterCounter is a literate translation of Example 3.3.8. (Actually, it is the other way round: I first developed the example in PVS and translated it later into LaTeX.) The proofs in PVS are straightforward.

[4] The definition of T_invariant uses PredT with an instantiation different from its definition. Therefore, the definition of T_invariant must stand in a different theory than the definition of PredT. I ignore these technical issues here.

[5] Enumeration types are syntactic sugar in PVS. They are mapped to an abstract datatype declaration with constant constructors.

```
InterCounter : Theory
Begin
  A : Type = { a1, a2, a3, a4 }
  B : Type = { b1, b2, b3, b4 }

  f : [A -> A] = Lambda(a:A) : IF a = a1 Then a1 Else a4 Endif
  g : [B -> B] = Lambda(b:B) : IF b = b1 Then b1 Else b4 Endif

  Importing TMorph

  c : [A -> TIface[A,A]] = Lambda(a:A) :
     Lambda(h : [A -> A]) : IF h = f Then a4 Else a1 Endif

  d : [B -> TIface[B,B]] = Lambda(b:B) :
     Lambda(k : [B -> B]) : IF k = g Then b4 Else b1 Endif

  R : PRED[[A,B]] = Lambda( a : A, b : B ) :
    (a=a1 And b=b1) OR (a=a2 And b=b2)

  S : PRED[[A,B]] = Lambda( a : A, b : B ) :
    (a=a1 And b=b1) OR (a=a3 And b=b3)

  R_bisim : Lemma T_bisimulation?(c,d)(R)
  S_bisim : Lemma T_bisimulation?(c,d)(S)

  RS_bisim : Lemma Not T_bisimulation?(c,d)( fib_and(R,S) )

  % now turn to invariants: use first projections of R and S from above
  P : PRED[A] = Lambda( a : A ) : a=a1 OR a=a2
  Q : PRED[A] = Lambda( a : A ) : a=a1 OR a=a3

  pq_proj : Lemma
     ( P = coprod(pi_1)(R) ) And ( Q = coprod(pi_1)(S) )

  Importing TInvariant

  P_inv : Lemma T_invariant?(c)(P)
  Q_inv : Lemma T_invariant?(c)(Q)

  PQ_inv : Lemma Not T_invariant?(c)( fib_and(P,Q) )
End InterCounter
```

Figure A.3.: PVS code for Example 3.3.8: The intersection of two T–bisimulations (invariants) is not a bisimulation (invariant) (file `intersection.pvs`).

As a last example I sketch the formalisation of Proposition 3.4.20 (about the equivalence of Aczel/Mendler and Hermida/Jacobs bisimulation on page 102). The PVS sources are from file `aczel.pvs`. The proof of this proposition proceeds by induction on the structure of the functor. As explained before, the induction itself cannot be formalised, but the induction steps can. First one has to formalise the following two inductive properties (copied from page 102 for convenience):

$$\begin{aligned} &\forall x \in G(X,X),\ y \in G(Y,Y),\ r \in G(R,R)\,. \\ &\qquad G(R,\pi_1)(r) = G(\pi_1,X)(x) \ \text{ and }\ G(R,\pi_2)(r) = G(\pi_2,Y)(y) \\ &\qquad\qquad \text{implies}\ \ \mathrm{Rel}(G)(R,R)(x,y) \end{aligned} \tag{A.1}$$

$$\begin{aligned} &\forall x \in G(X,X),\ y \in G(Y,Y), \\ &\qquad \mathrm{Rel}(G)(R,R)(x,y)\ \text{ implies }\ \exists r \in G(R,R)\,. \\ &\qquad\qquad G(R,\pi_1)(r) = G(\pi_1,X)(x)\ \text{ and }\ G(R,\pi_2)(r) = G(\pi_2,Y)(y) \end{aligned} \tag{A.2}$$

These two properties use both the object part and the morphism part of G. Consider the term $G(R,\pi_1)(r)$ that occurs in (A.1). In the different induction steps one has to use a different type for r: In the induction step for $G_1 \times G_2$ it is of product type, while in the step for $G_1 \Rightarrow G_2$ it is of function type. In the same way the induction steps use different functions for $G(R,\pi_1)$. Therefore I decided to abstract away all functor applications (both to objects and to morphisms) from the properties (A.1) and (A.2). For the object $G(X,X)$ I substitute by the type TXX, for the object $G(R,R)$ the type TRR, and so on. The same happens for functions: for instance $G(R,\pi_1)$ becomes TRpi_1 : [TRR –> TRX]. The predicate lifting $\mathrm{Rel}(G)(R,R)$ becomes the predicate RelT : PRED[[TXX,TYY]]. With this translation the two properties read as follows:

```
ast_property : bool =
  Forall(xx : TXX, yy : TYY) : Forall(r : TRR) :
    TRpi_1(r) = Tpi_1X(xx) And TRpi_2(r) = Tpi_2Y(yy)
      Implies  RelT( xx, yy )

dagger_property : bool =
  Forall(xx : TXX, yy : TYY) : RelT(xx,yy)     Implies
      Exists(rr : TRR) :
        Tpi_1X(xx) = TRpi_1(rr) And
        Tpi_2Y(yy) = TRpi_2(rr)
```

I further decided to declare all the abstracted items (i.e., all the TXX, TRpi_1, ...) as theory parameters. For symmetry I add the two types TXR and TYR standing for $G(X,R)$ and $G(Y,R)$ (which would not be needed to express the two properties). I also complete the functions and add the missing projections, like for instance TXpi_1 : [TXR –> TXX], which stands for $G(X,\pi_1)$. This makes altogether seven type parameters and nine value parameters for the theory InductionProperties that contains the definitions of ast_property and dagger_property:

```
InductionProperties[TXX, TYY, TRR, TXR, TYR, TRX, TRY : Type,
        RelT : PRED[[TXX,TYY]],
        TXpi_1 : [TXR -> TXX], Tpi_1R : [TXR -> TRR],
        Tpi_1X : [TXX -> TRX], TRpi_1 : [TRR -> TRX],
        TYpi_2 : [TYR -> TYY], Tpi_2R : [TYR -> TRR],
        Tpi_2Y : [TYY -> TRY], TRpi_2 : [TRR -> TRY]
    ] : Theory
```

A basic property that holds for functors is that

$$G(\pi_1, X) \circ G(X, \pi_1) \quad = \quad G(\pi_1, \pi_1) \quad = \quad G(R, \pi_1) \circ G(\pi_1, R) \qquad \text{(A.3)}$$

(and similarly for the second projection). Therefore I add the following assuming clause to the theory InductionProperties:

```
Assuming
  pi_1_commutes : Assumption  Tpi_1X o TXpi_1 = TRpi_1 o Tpi_1R
  pi_2_commutes : Assumption  Tpi_2Y o TYpi_2 = TRpi_2 o Tpi_2R
Endassuming
```

Inside of theory InductionProperties the two assumptions have the status of axioms: One can use them in proofs without the requirement to prove the assumptions themselves. From the outside the assumptions appear as additional properties that are required for all instantiations. For every use of the theory InductionProperties the user has to prove two type correctness conditions. The two TCC's are generated from the two assumptions by substituting the actual theory instantiations for the parameters.

The next step in the proof of Proposition 3.4.20 is the induction that shows that the two properties (A.1) and (A.2) hold for all extended polynomial functors. Consider the case $G(Y, X) = A$ for a constant set A. For the proof in PVS one has to instantiate the 16 parameters of InductionProperties appropriately. All the type parameters are instantiated by A, because $G(X, X) = A$. The relation lifting RelT is instantiated with the equality relation over A. Finally, all the functions are instantiated with the identity function, because $G(f, g) = \mathrm{id}_A$. The theory that formalises the case $G(Y, X) = A$ is

```
Const[ A : Type] : Theory   % G(Y,X) = A (constant)
Begin
  Importing InductionProperties[A,A,A,A,A,A,A, equality[A],
        id[A],id[A],id[A],id[A],id[A],id[A],id[A],id[A] ]

  ast_const : Lemma ast_property
  dagger_const : Lemma dagger_property
End Const
```

The proofs of the two lemmas ast_const and dagger_const is trivial. Also the two TCC's that are generated for the importing are easy.

The induction step for the case $G(Y, X) = G_1(Y, X) \times G_2(Y, X)$ is much more complicated. It is formalised in the theory Prod in Figure A.4. In the induction step we need to assume that the properties (A.1) and (A.2) hold for both G_1 and G_2. I solve this by two sets of theory parameters, one for G_1 and one for G_2. The parameters for G_1 are called T1XX, RelT1, T1Rpi_1, and so on. For G_2 they are T2XX, RelT2, T2Rpi_1. Of course I need now assumptions that state the Equation (A.3) for G_1 and G_2. This explains the first 20 lines of Figure A.1.

The lemma ast_prod in theory Prod assumes that the ast_property holds for the type parameters that simulate G_1 and for those that simulate G_2. It then states on line 31 that the property also holds for an instantiation of ast_property that simulates $G_1 \times G_2$. This latter instantiation is straightforward, for instance $G(X, X) = G_1(X, X) \times G_2(X, X)$ accounts for the first argument [T1XX, T2XX] on line 31. And from $G(\pi_1, X) = G_1(\pi_1, X) \times G_2(\pi_1, X)$ it follows the first argument times(T1pi_1X, T2pi_1X) on line 35. The theory Prod contains a similar lemma dagger_prod for A.2.

The induction step for coproduct is very similar to what I just showed. The induction step for the exponent $G(Y, X) = G_1(A, Y) \Rightarrow G_2(Y, X)$ shows two interesting points. The first one is that the restriction of extended polynomial functors must appear there somehow. It turned out that for the completion of the proofs it is enough to exploit the fact that some of the functions that stand for the action of G_1 are always bijections (because they collapse to identities). Let $T(X, Y) = G_1(A, Y)$ (for an extended polynomial functor G_1) then $T(\pi_1, R) = \mathrm{id}_{G_1(A,R)}$ is clearly bijective. In the theory for the induction step I have therefore the additional assumption

```
polynomial : Assumption
    bijective?(T1pi_1R) and bijective?(T1pi_1X) and
    bijective?(T1pi_2Y) and bijective?(T1pi_2R)
```

The second interesting point is that in the assumptions for G_1 one has to exchange the inductive properties: In order to prove ast_property for $G_1 \Rightarrow G_2$ one has to assume dagger_property for G_1 and ast_property for G_2.

This finishes the formalisation of the induction for extended polynomial functors. The PVS source code contains an additional theory that proves that the two inductive properties (A.1) and (A.2) do indeed imply Proposition 3.4.20. I would like to skip this material here.

I should note that in the development of the proof that I just described the PVS formalisation was extremely valuable. The attempt the formalise the induction steps in PVS invalidated two proofs that have been traditionally developed with pencil and paper. Further, the careful investigation of the proof goals that I got without the preceding assumption polynomial lead to the discovery of the Examples 3.3.11 and 3.3.12 (on page 91ff).

```
Prod[ T1XX, T1YY, T1RR, T1XR, T1YR, T1RX, T1RY : Type, % parameters for G1
      RelT1 : PRED[[T1XX,T1YY]],
      T1Xpi_1 : [T1XR -> T1XX], T1pi_1R : [T1XR -> T1RR],
      T1pi_1X : [T1XX -> T1RX], T1Rpi_1 : [T1RR -> T1RX],
      T1Ypi_2 : [T1YR -> T1YY], T1pi_2R : [T1YR -> T1RR],
      T1pi_2Y : [T1YY -> T1RY], T1Rpi_2 : [T1RR -> T1RY],

      T2XX, T2YY, T2RR, T2XR, T2YR, T2RX, T2RY : Type, % parameters for G2
      RelT2 : PRED[[T2XX,T2YY]],
      T2Xpi_1 : [T2XR -> T2XX], T2pi_1R : [T2XR -> T2RR],
      T2pi_1X : [T2XX -> T2RX], T2Rpi_1 : [T2RR -> T2RX],
      T2Ypi_2 : [T2YR -> T2YY], T2pi_2R : [T2YR -> T2RR],
      T2pi_2Y : [T2YY -> T2RY], T2Rpi_2 : [T2RR -> T2RY]
      ] : Theory
Begin
  Assuming
    T1pi_1_commutes : Assumption T1pi_1X o T1Xpi_1 = T1Rpi_1 o T1pi_1R
    T1pi_2_commutes : Assumption T1pi_2Y o T1Ypi_2 = T1Rpi_2 o T1pi_2R
    T2pi_1_commutes : Assumption T2pi_1X o T2Xpi_1 = T2Rpi_1 o T2pi_1R
    T2pi_2_commutes : Assumption T2pi_2Y o T2Ypi_2 = T2Rpi_2 o T2pi_2R
  Endassuming
  Importing InductionProperties

  ast_prod : Lemma
    ast_property[T1XX, T1YY, T1RR, T1XR, T1YR, T1RX, T1RY, RelT1,
                 T1Xpi_1, T1pi_1R, T1pi_1X, T1Rpi_1,
                 T1Ypi_2, T1pi_2R, T1pi_2Y, T1Rpi_2]
        And
    ast_property[T2XX, T2YY, T2RR, T2XR, T2YR, T2RX, T2RY, RelT2,
                 T2Xpi_1, T2pi_1R, T2pi_1X, T2Rpi_1,
                 T2Ypi_2, T2pi_2R, T2pi_2Y, T2Rpi_2]
        Implies
    ast_property[[T1XX,T2XX], [T1YY,T2YY], [T1RR,T2RR], [T1XR,T2XR],
                 [T1YR,T2YR], [T1RX,T2RX], [T1RY,T2RY],
                 relprod(RelT1,RelT2),
                 times(T1Xpi_1, T2Xpi_1), times(T1pi_1R, T2pi_1R),
                 times(T1pi_1X, T2pi_1X), times(T1Rpi_1, T2Rpi_1),
                 times(T1Ypi_2, T2Ypi_2), times(T1pi_2R, T2pi_2R),
                 times(T1pi_2Y, T2pi_2Y), times(T1Rpi_2, T2Rpi_2)]
```

Figure A.4.: PVS code for the induction step $G(Y,X) = G_1(Y,X) \times G_2(Y,X)$ of the proof of Proposition 3.4.20 (file `aczel.pvs`).

This section presented a few examples from the PVS formalisation of Section 2.4 and Chapter 3. The translation of fibred category theory follows the simple approach of translating objects and morphisms (in the base) to types and functions (in PVS), respectively. The preceding examples of PVS source code show that with this approach one can express and prove a lot of important properties in PVS in a very natural way. Properties that involve a quantification over a class of functors cannot be modelled. However, the last pages show that, with some experience in PVS, it is possible to formalise substantial parts of the proofs of such properties.

A.2. Contents of the PVS Formalisation

The following tables relates the examples and results of the present thesis with the theorems and theories of the PVS source.

Lemma /	Location in the PVS sources		
Example	file	theory	lemma
2.4.5 (1)	fibprops	PredProps	mon_prod, mon_coprod, mon_exp
2.4.5 (2)	fibprops	RelProps	mon_prod, mon_coprod, mon_exp
2.4.6 (1)	fibrations	Bc, BCRel	bc, bc_rel
2.4.6 (2)	fibrations	Frobenius, FrobeniusRel	frobenius frobenius_rel
2.4.7	fibprops	PredProps	truth_prod, truth_coprod, truth_exp
2.4.8	fibprops	RelProps	eq_prod, eq_coprod, eq_exp
2.4.9 (1)	fibprops	PredProps	coll_and_prod, coll_and_coprod, coll_and_exp_pol, coll_and_exp
2.4.9 (2)	fibprops	RelProps	coll_and_prod, coll_and_coprod, coll_and_exp_pol, coll_and_exp
2.4.10 (1)	fibprops	PredProps	coll_or_prod, coll_or_coprod, coll_or_exp_pol
2.4.10 (2)	fibprops	RelProps	coll_or_prod, coll_or_coprod, coll_or_exp_pol
2.4.11	fibprops	RelPropsOp	op_prod, op_coprod, op_exp
2.4.12	fibprops	RelProps	comp_prod, comp_coprod, comp_exp_pol, comp_exp_left

Lemma /	Location in the PVS sources		
Example	file	theory	lemma
2.4.13	fibprops	BisimProjProp	times, sum, exp_pol, exp_left
2.4.14	fibprops	CapInvProp	times, sum, exp, exp_pol, exp_nonempty
2.4.15 (1)	fibprops	FibProps	fibred_product, fibred_coproduct, fibred_exp
2.4.15 (2)	fibprops	FibRel	rel_prod_fib, rel_coprod_fib, rel_exp_fib
2.4.16	exp	ExpCounter2	
2.4.17 (1)	fibprops	FibProps	cofibred_product, cofibred_coproduct
2.4.17 (1)	fibprops	CofibRel	rel_prod_cofib, rel_coprod_cofib
2.4.17 (2)	fibprops	FibProps	const_exp1
2.4.17 (3)	fibprops	CofibRel	rel_exp_cofib1
Eq. 2.7	fibrations	PredCont	graf, graf2
Eq. 2.7	fibrations	BCfor_graph	
Eq. 3.1	per	PerUnion	percl_rel_exp_pol_coll
3.3.7 (1)	intersection	InterCounter	
3.3.7 (1)	union	UnionBisim	
3.3.7 (2)	relcomp	BisimComp	
3.3.7 (3)	graph	GraphCounter	
3.3.7 (4)	image_morph	Image_Morph_counter	
3.3.7 (5)	mendler_counter		
3.3.7 (6)	inv-char	InvCharCounter	
3.3.7 (7)	invariant	BisimProjCounter	
3.3.7 (8)	invariant	BisimCapInvCounter	
3.3.7 (9)	kernel	KernelCounter	

Lemma /	Location in the PVS sources		
Example	file	theory	lemma
3.3.8	t		
3.3.8	intersection	InterCounter	
3.3.9	union	UnionBisim	
3.3.11	aczel_counter3	Fun_not_T	
3.3.12	aczel_counter3	Bisim_not_T	
3.4.10	invariant	BisimProj	bisim_proj_const, bisim_proj_id, bisim_proj_times, bisim_proj_plus, bisim_proj_exp, bisim_proj_exp_epf
3.4.12	invariant	BisimProjCounter	
3.4.13	invariant	BisimCapInv	bisim_cap_inv_const, bisim_cap_inv_id, bisim_cap_inv_prod, bisim_cap_inv_coprod, bisim_cap_inv_exp, bisim_cap_inv_exp_epf
3.4.15	invariant	BisimCapInvCounter	
3.4.17	image_morph	Image_Morph_counter	
3.4.19	inv–char	InvCharCounter	
3.4.20	aczel		
3.4.29	per	PReflexive	prefl_union
3.5.2 (1)	per	PerLift	per_eq, per_sum, per_prod, per_exp
3.5.3	per	DomainLift	domain_sum, domain_prod, domain_exp_pol
3.5.4 (1)	per	PerLift	per_cl_sum
3.5.4 (2)	per	PerLift	per_cl_prod
3.5.5 (1)	per	PerUnion	percl_rel_prod_coll
3.5.5 (2)	per	PerUnion	percl_rel_sum_coll
Eq. 3.1	per	PerUnion	percl_rel_exp_pol_coll

Lemma /	Location in the PVS sources		
Example	file	theory	lemma
3.5.8	per_extended		
3.5.10	perlift_counter	PerBisimCounter2	
3.6.1	union	KNoFinal	
3.7.1 (1)	rellist	PredList	predlist_mon
3.7.1 (1)	rellist	RelList	rellist_mon
3.7.1 (2)	rellist	FibPredList, FibRelList	
3.7.1 (3)	rellist	PredListProps, RelListProps, CapInvProj	
3.7.1 (4)	rellist	RelListPer	
4.2.2 (1)	variance	Variance	join_well, subst_well
4.2.2 (2)	variance	Variance	subst_comm, subst_asso, subst_zero
4.2.2 (3)	variance	Variance	join_comm, join_asso, join_id
4.2.2 (4)	variance	Variance	distrib
4.4.6	Queue_model	QueueModel	
4.5.10	Queue_model	QueueModal	

B. The CCSL Grammar

This appendix contains the complete CCSL grammar. It is given in a BNF–like notation. Brackets [. . .] denote optional components, braces ⦃. . .⦄ denote arbitrary repetition (including zero times), and parenthesis (. . .) denote grouping. Terminals are set in `UPPERCASE TYPEWRITER`, non–terminals in *lowercase slanted*. The terminal symbols for parenthesis and brackets are written as (,), [, and].

```
file                 ::=  ⦃ declaration ⦄ EOF
declaration          ::=  classspec
                      |   adtspec
                      |   groundsignature
                      |   typedef
                      |   groundtermdef
classspec            ::=  BEGIN identifier [ parameterlist ] :
                          [ FINAL ] CLASSSPEC
                          ⦃ importing ⦄ ⦃ classsection ⦄
                          END identifier
parameterlist        ::=  [ parameters ⦃ , parameters ⦄ ]
parameters           ::=  identifier ⦃ , identifier ⦄ : [ variance ] TYPE
variance             ::=  POS
                      |   NEG
                      |   MIXED
                      |   ( numberorquestion , numberorquestion )
numberorquestion     ::=  ?
                      |   number
classsection         ::=  inheritsection
                      |   [ visibility ] attributesection [ ; ]
                      |   [ visibility ] methodsection [ ; ]
                      |   definitionsection
                      |   classconstructorsection [ ; ]
                      |   assertionsection
                      |   creationsection
                      |   theoremsection
                      |   requestsection [ ; ]
```

```
visibility                 ::=  PUBLIC
                             |  PRIVATE
inheritsection             ::=  INHERIT FROM ancestor ⦃ , ancestor ⦄
ancestor                   ::=  identifier [ argumentlist ]
                                [ RENAMING renaming ⦃ AND renaming ⦄ ]
renaming                   ::=  identifier AS identifier
attributesection           ::=  ATTRIBUTE member ⦃ ; member ⦄
methodsection              ::=  METHOD member ⦃ ; member ⦄
member                     ::=  identifier : type -> type
definitionsection          ::=  DEFINING member formula ; ⦃ member formula ; ⦄
classconstructorsection    ::=  CONSTRUCTOR classconstructor ⦃ ; classconstructor ⦄
classconstructor           ::=  identifier : type
                             |  identifier : type -> type
assertionsection           ::=  ASSERTION ⦃ importing ⦄
                                [ assertionselfvar ] ⦃ freevarlist ⦄
                                namedformula ⦃ namedformula ⦄
assertionselfvar           ::=  SELFVAR identifier : SELF
freevarlist                ::=  VAR vardecl ⦃ ; vardecl ⦄
creationsection            ::=  CREATION ⦃ importing ⦄ ⦃ freevarlist ⦄
                                namedformula ⦃ namedformula ⦄
namedformula               ::=  identifier : formula ;
requestsection             ::=  REQUEST request ⦃ ; request ⦄
request                    ::=  identifier : type
theoremsection             ::=  THEOREM ⦃ importing ⦄ ⦃
                                freevarlist ⦄ namedformula ⦃ namedformula ⦄
formula                    ::=  FORALL ( vardecl ⦃ , vardecl ⦄ ) ( : | . ) formula
                             |  EXISTS ( vardecl ⦃ , vardecl ⦄ ) ( : | . ) formula
                             |  LAMBDA ( vardecl ⦃ , vardecl ⦄ ) ( : | . ) formula
                             |  LET binding ⦃ ( ; | , ) binding ⦄ [ ; | , ] IN formula
                             |  formula IFF formula
                             |  formula IMPLIES formula
                             |  formula OR formula
                             |  formula AND formula
                             |  IF formula THEN formula ELSE formula
```

```
                        |  NOT formula
                        |  formula infix_operator formula
                        |  ALWAYS formula FOR
                           [ identifier [ argumentlist ] : : ] methodlist
                        |  EVENTUALLY formula FOR
                           [ identifier [ argumentlist ] : : ] methodlist
                        |  CASES formula OF caselist [ ; | , ] ENDCASES
                        |  formula WITH [ update ⦃ , update ⦄ ]
                        |  formula . qualifiedid
                        |  formula formula
                        |  TRUE
                        |  FALSE
                        |  PROJ_N
                        |  number
                        |  qualifiedid
                        |  ( formula : type )
                        |  ( formula ⦃ , formula ⦄ )
vardecl               ::=  identifier ⦃ , identifier ⦄ : type
methodlist            ::=  { identifier ⦃ , identifier ⦄ }
qualifiedid           ::=  idorinfix
                        |  identifier [ argumentlist ] : : idorinfix
idorinfix             ::=  ( infix_operator )
                        |  identifier
binding               ::=  identifier [ : type ] = formula
caselist              ::=  pattern : formula ⦃ ( ; | , ) pattern : formula ⦄
pattern               ::=  identifier [ ( identifier ⦃ , identifier ⦄ ) ]
update                ::=  formula := formula
adtspec               ::=  BEGIN identifier [ parameterlist ] : ADT
                           ⦃ adtsection ⦄
                           END identifier
adtsection            ::=  adtconstructorlist [ ; ]
adtconstructorlist    ::=  CONSTRUCTOR adtconstructor ⦃ ; adtconstructor ⦄
adtconstructor        ::=  identifier [ adtaccessors ] : type
                        |  identifier [ adtaccessors ] : type -> type
adtaccessors          ::=  ( identifier ⦃ , identifier ⦄ )
```

groundsignature	::=	`BEGIN` *identifier* [*parameterlist*] `:` `GROUNDSIGNATURE` ⦃ *importing* ⦄ ⦃ *signaturesection* ⦄ `END` *identifier*
signaturesection	::=	*typedef*
	|	*signaturesymbolsection* [`;`]
signaturesymbolsection	::=	`CONSTANT` *termdef* ⦃ `;` *termdef* ⦄
typedef	::=	`TYPE` *identifier* [*parameterlist*] [`=` *type*]
groundtermdef	::=	`CONSTANT` *termdef* [`;`]
termdef	::=	*idorinfix* [*parameterlist*] `:` *type* [*formula*]
type	::=	`SELF`
	|	`CARRIER`
	|	`BOOL`
	|	`[` *type* ⦃ `,` *type* ⦄ `->` *type* `]`
	|	`[` *type* ⦃ `,` *type* ⦄ `]`
	|	*qualifiedid*
	|	*identifier argumentlist*
argumentlist	::=	`[` *type* ⦃ `,` *type* ⦄ `]`
importing	::=	`IMPORTING` *identifier* [*argumentlist*]

CCSL Keywords

The following words are reserved.

adt always and as assertion
attribute begin bool carrier cases
classspec constant constructor creation defining
else end endcases eventually exists
false final for forall from
groundsignature if iff implies importing
in inherit lambda let method
mixed neg not of or
pos private public renaming request
self selfvar then theorem true
type var with

Include Directive

The include directive has the following form:

$$\mathit{include} \quad ::= \quad \texttt{\#include "}\mathit{string}\texttt{"}$$

The string is interpreted as a file name. The compiler substitutes the contents of the file for the include directive. The directive can stand at any place in the input.

C. Bibliography

Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag, New York.

Aczel, P. and P. F. Mendler (1989). *A final coalgebra theorem*. In Pitt, D. H., D. E. Rydeheard, P. Dybjer, A. M. Pitts, and A. Poigné, eds.: *Proceedings of the Conference on Category Theory and Computer Science*, vol. 389 of *Lecture Notes in Computer Science*, pp. 357–365, Berlin. Springer.

Adámek, J., H. Herrlich, and G. E. Strecker (1990). *Abstract and Concrete Categories*. Pure and applied mathematics. John Wiley and Sons, New York, New York.

Aho, A. V., R. Sethi, and J. D. Ullman (1986). *Compilers: principles, techniques, tools*. Addison-Wesley.

Alpern, Bowen and F. B. Schneider (1985). *Defining liveness*. Information Processing Letters, 21(4):181–185.

Astesiano, E., A. Evans, R. France, G. Geniloud, M. Gogolla, B. Henderson-Sellers, J. Howse, H. Hussmann, S. Iida, S. Kent, A. L. Guennec, T. Mens, R. Mitchell, O. Radfelder, G. Reggio, M. Richters, B. Rumpe, P. Stevens, K. van den Berg, P. van den Broek, and R. Wieringa (1999). *UML semantics FAQ*. In Moreira, Ana and S. Demeyer, eds.: *Object-Oriented Technology. ECOOP'99 Workshop Reader*, vol. 1743 of *Lecture Notes in Computer Science*, pp. 33–56. Springer-Verlag.

Augustsson, L., D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler (1999). *Report on the Programming Language Haskell 98*. Available via http://www.haskell.org/.

Back, R.-J. and J. von Wright (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

Barendregt, H. P. (1992). *Lambda calculi with types*. In Abramsky, S., D. M. Gabbay, and T. Maibaum, eds.: *Handbook of Logic in Computer Science*, vol. 2. Oxford Science Publications.

Barr, M. and C. Wells (1995). *Category Theory for Computing Science*. Prentice Hall Intl. Series in Computer Science. Prentice Hall, London, 2nd ed. (1st ed., New York, 1990).

Bénabou, J. (1985). *Fibered categories and the foundations of naive category theory.* Journal of Symbolic Logic, 50(1):10–37.

Berg, J. van den, M. Huisman, B. Jacobs., and E. Poll (1999). *A type-theoretic memory model for verification of sequential Java programs.* In Bert, Didier, C. Choppy, and P. Mosses, eds.: *Recent Trends in Algebraic Development Techniques, WADT '99*, vol. 1827 of *Lecture Notes in Computer Science*, pp. 1–21. Springer-Verlag.

Berg, J. van den and B. Jacobs (2001). *The LOOP compiler for Java and JML.* In Margaria, T. and W. Yi, eds.: *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 2031 of *Lecture Notes in Computer Science*, pp. 299–312.

Berghofer, S. and M. Wenzel (1999). *Inductive datatypes in HOL — lessons learned in formal-logic engineering.* In Bertot, Y., G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, eds.: *Theorem Proving in Higher Order Logics*, vol. 1690 of *Lecture Notes in Computer Science*, pp. 19–36. Springer.

Börger, E. (2002). *The origins and the development of the ASM method for high level system design and analysis.* Journal of Universal Computer Science, 8(1):2–74.

Bruce, K., L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce (1995). *On binary methods.* Theory and Practice of Object Systems, 1(3):221–242.

Bruce, K. B., L. Cardelli, and B. C. Pierce (1997). *Comparing object encodings.* In Abadi, M. and T. Ito, eds.: *Theoretical Aspects of Computer Software (TACS), Sendai, Japan.*

Bruijn, N. G. de (1972). *Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem.* Indagationes Mathematicae, 34(5):381–392.

Bruns, G. and S. Anderson (1994). *The formalization and analysis of a communications protocol.* Formal Aspects of Computing, 6(1):92–112.

Cardelli, Luca and P. Wegner (1985). *On understanding types, data abstraction, and polymorphism.* ACM Computing Surveys, 17(4):471–522.

Castagna, Giuseppe (1997). *Object-Oriented Programming: A Unified Foundation.* Progress in Theoretical Computer Science. Birkhauser, Boston.

Chen, Z. (2000). *Java Card Technology for Smart Cards.* The Java Series. Addison-Wesley.

Cîrstea, C. (1999). *A coalgebraic equational approach to specifying observational structures.* In Jacobs, B. and J. Rutten, eds.: *Coalgebraic Methods in Computer Science '99*, vol. 19 of *ENTCS*. Elsevier, Amsterdam.

Clark, T., A. Evans, and S. Kent (2001). *The metamodelling language calculus: Foundation semantics for UML.* In Hussmann, Heinrich, ed.: *Fundamental Approaches to Software Engineering 2001, Proceedings*, vol. 2029 of *Lecture Notes in Computer Science*, pp. 17–31. Springer.

Cockett, J. R. B. and D. Spencer (1992). *Strong categorical datatypes I*. In Seely, R. A. G., ed.: *Proc. of Int. Summer Category Theory Meeting, Montréal, Québec, 23–30 June 1991*, vol. 13 of *Canadian Mathematical Society Conf. Proceedings*, pp. 141–169. American Mathematical Society, Providence, RI.

Cockett, R. and T. Fukushima (1992). *About Charity*. Yellow Series Report 92/480/18, Department of Computer Science, University of Calgary.

Cook, W. R. (1989). *A proposal for making Eiffel type-safe.* The Computer Journal, 32(4):305–311.

Cook, W. R., W. Hill, and P. S. Canning (1990). *Inheritance is not subtyping.* In ACM, ed.: *POPL '90. Proceedings of the seventeenth annual ACM symposium on Principles of programming languages*, pp. 125–135, New York, NY, USA. ACM Press.

Corradini, A. (1998). *A completeness result for equational deduction in coalgebraic specification.* In Presicce, F. Parisi, ed.: *Recent Trends in Data Type Specification*, vol. 1376 of *Lecture Notes in Computer Science*, pp. 190–205. Springer, Berlin.

Corradini, A., M. Lenisa, and U. Montanari, eds. (2001). *Workshop on Coalgebraic Methods in Computer Science '01*, vol. 44 of *ENTCS*, Genova. Elsevier.

Crow, J., S. Owre, J. Rushby, N. Shankar, and M. Srivas (1995). *A tutorial introduction to PVS*. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida. Computer Science Laboratory, SRI International.

Diaconescu, R. and K. Futatsugi (1998). *CafeOBJ Report: the Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification.* World Scientific, Singapore.

Eiffel FAQ (2001). *Eiffel: Frequently asked questions.* available at ftp://rtfm.mit.edu/pub/usenet/news.answers/eiffel-faq.

Fowler, M. (1999). *UML distilled: a brief guide to the standard object modeling language.* Addison–Wesley. 2nd edition.

Goguen, J. and G. Malcolm (1996). *Algebraic Semantics of Imperative Programs.* MIT Press, Cambridge, Mass., 1 ed.

Goguen, J. and G. Malcolm (2000). *A hidden agenda*. Theoretical Computer Science, 245(1):55–101.

Goldblatt, R. (1992). *Logics of Time and Computation, Second Edition, Revised and Expanded*, vol. 7 of *CSLI Lecture Notes*. CSLI, Stanford.

Goldblatt, R. (2001a). *A calculus of terms for coalgebras of polynomial functors*. In Corradini, A., M. Lenisa, and U. Montanari, eds.: *Coalgebraic Methods in Computer Science '01*, vol. 44 of *ENTCS*. Elsevier, Amsterdam.

Goldblatt, R. (2001b). *What is the coalgebraic analogue of birkhoff's variety theorem*. To appear in Theoretical Computer Science.

Gordon, M. J. C. and T. F. Melham (1993). *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press.

Gosling, J., B. Joy, and G. Steele (1996). *The Java Language Specification*. Addison-Wesley.

Griffioen, D. and M. Huisman (1998). *A comparison of PVS and Isabelle/HOL*. In Grundy, J. and M. Newey, eds.: *Theorem Proving in Higher Order Logics*, vol. 1479 of *Lecture Notes in Computer Science*, pp. 123–142. Springer, Berlin.

Gumm, H. P. (1999). *Elements of the general theory of coalgebras*. Preliminary version, available at http://www.mathematik.uni-marburg.de/∼gumm/Papers/publ.html.

Gunter, Elsa L. (1992). *Why we can't have SML style* `datatype` *declarations in HOL*. In Claese, L. J. M. and M. J. C. Gordon, eds.: *Higher Order Logic Theorem Proving and Its Applications*, vol. A–20 of *IFIP Transactions*, pp. 561–568. North-Holland Press.

Härtig, H., R. Baumgartl, M. Borriss, C.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter (1998). *DROPS: OS support for distributed multimedia applications*. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal.

Hennicker, R. and M. Bidoit (1999). *Observational logic*. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98)*, vol. 1548 of *Lecture Notes in Computer Science*, pp. 263–277. Springer–Verlag.

Hennicker, R. and A. Kurz (1999). (Ω, Ξ)*–Logic: On the algebraic extension of coalgebraic specifications*. In Jacobs, B. and J. Rutten, eds.: *Coalgebraic Methods in Computer Science '99*, vol. 19 of *ENTCS*, pp. 195–212. Elsevier.

Hensel, U. (1999). *Definition and Proof Principles for Data and Processes*. PhD thesis, Univ. of Dresden, Germany.

Hensel, U., M. Huisman, B. Jacobs, and H. Tews (1998). *Reasoning about classes in object–oriented languages: Logical models and tools*. In Hankin, Ch., ed.: *European Symposium on Programming*, vol. 1381 of *Lecture Notes in Computer Science*, pp. 105–121. Springer, Berlin.

Hensel, U. and B. Jacobs (1997). *Proof principles for datatypes with iterated recursion*. In Moggi, E. and G. Rosolini, eds.: *Category Theory and Computer Science*, vol. 1290 of *Lecture Notes in Computer Science*, pp. 220–241. Springer, Berlin.

Hermida, C. and B. Jacobs (1998). *Structural induction and coinduction in a fibrational setting*. Information and Computation, 145(2):107–152.

Hohmuth, M. (1998). *The Fiasco kernel: Requirements definition*. Technical Report TUD–FI–12, TU Dresden. Available at URL: http://os.inf.tu-dresden.de/fiasco/doc.html.

Hohmuth, M. (2000). *The Fiasco kernel: System architecture*. Available at URL: http://os.inf.tu-dresden.de/fiasco/doc.html.

Hudak, P., J. Peterson, and J. H. Fasel (1992). *A Gentle Introduction to Haskell 98*. Available via http://haskell.cs.yale.edu/tutorial/.

Hughes, Jesse (2001). *Modal operators for coequations*. In Corradini, A., M. Lenisa, and U. Montanari, eds.: *Coalgebraic Methods in Computer Science '01*, vol. 44 of *ENTCS*. Elsevier, Amsterdam.

Huisman, M. (2001). *Reasoning about Java programs in Higher-order logic using PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands.

Huisman, M. and B. Jacobs (2000). *Inheritance in higher order logic: Modeling and reasoning*. In Aagaard, M. and J. Harrison, eds.: *Theorem Proving in Higher Order Logics*, vol. 1869 of *Lecture Notes in Computer Science*, pp. 301–319. Springer, Berlin.

Intel (1999). *Intel Architecture Software Developer's Manual, Volume 3: System Programming*. Intel Corp.

Jacobs, B. (1995). *Mongruences and cofree coalgebras*. In Alagar, V.S. and M. Nivat, eds.: *Algebraic Methodology and Software Technology*, vol. 936 of *Lecture Notes in Computer Science*, pp. 245–260. Springer, Berlin.

Jacobs, B. (1996a). *Inheritance and cofree constructions*. In Cointe, P., ed.: *Proceedings ECOOP '96*, vol. 1098 of *Lecture Notes in Computer Science*, pp. 210–231. Springer-Verlag.

Jacobs, B. (1996b). *Objects and classes, co–algebraically.* In Freitag, B., C. Jones, C. Lengauer, and H.-J. Schek, eds.: *Object–Orientation with Parallelism and Peristence*, pp. 83–103. Kluwer Acad. Publ.

Jacobs, B. (1997a). *Behaviour-refinement of coalgebraic specifications with coinductive correctness proofs.* In Bidoit, M. and M. Dauchet, eds.: *TAPSOFT'97: Theory and Practice of Software Development*, vol. 1214 of *Lecture Notes in Computer Science*, pp. 787–802. Springer, Berlin.

Jacobs, B. (1997b). *Invariants, bisimulations and the correctness of coalgebraic refinements.* In Johnson, M., ed.: *Algebraic Methodology and Software Technology*, vol. 1349 of *Lecture Notes in Computer Science*, pp. 276–291. Springer, Berlin.

Jacobs, B. (1999a). *Categorical Logic and Type Theory*, vol. 141 of *Studies in Logic and the Foundations of Mathematics.* North Holland, Elsevier.

Jacobs, B. (1999b). *The temporal logic of coalgebras via Galois algebras.* Techn. Rep. CSI-R9906, Comput. Sci. Inst., Univ. of Nijmegen.

Jacobs, B. (2000). *Many-sorted coalgebraic modal logic: a model-theoretic study.* Technical Report CSI-R0020, University of Nijmegen.

Jacobs, B. (2002). *Exercises in coalgebraic specification.* In R. Backhouse, R. Crole and J. Gibbons, eds.: *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, vol. 2297 of *Lecture Notes in Computer Science*, pp. 237–280. Springer, Berlin.

Jacobs, B., J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews (1998a). *Reasoning about classes in Java (preliminary report).* In *Object–Oriented Programming, Systems, Languages and Applications*, pp. 329–340. ACM Press.

Jacobs, B., L. Moss, H. Reichel, and J. Rutten, eds. (1998b). *Workshop on Coalgebraic Methods in Computer Science '98*, vol. 11 of *ENTCS*, Lisbon, Portugal. Elsevier.

Jacobs, B. and J. Rutten (1997). *A tutorial on (co)algebras and (co)induction.* EATCS Bulletin, 62:222–259.

Jacobs, B. and J. Rutten, eds. (1999). *Workshop on Coalgebraic Methods in Computer Science '99*, vol. 19 of *ENTCS*, Amsterdam. Elsevier. Available at URL http://www.elsevier.nl/cas/tree/store/tcs/free/noncas/pc/volume19.htm.

Jacobs, B. and H. Tews (2001). *Assertional and behavioural refinement in coalgebraic specification.* Submitted.

Jay, C. B. (1996). *Data categories*. In Houle, M. E. and P. Eades, eds.: *Proceedings of Conference on Computing: The Australian Theory Symposium*, pp. 21–28. Australian Computer Science Communications.

Kawahara, Y. and M. Mori (2000). *A small final coalgebra theorem*. Theoretical Computer Science, 233(1–2):129–145.

Kellomäki, P. (1997). *Mechanical Verification of Invariant Properties of DisCo Specifications*. PhD thesis, Tampere University of Technology, Finland.

Kobryn, C. (1999). *UML 2001: a standardization odyssey*. Communications of the ACM, 42(10):29–37.

Kock, E. A. de and G. Essink (1999). *Y–chart application programmer's interface*. Technical note 0008/99, Philips Naturkundig Laboratorium.

Kurz, A. (1998). *Specifying coalgebras with modal logic*. In Jacobs, B., L. Moss, H. Reichel, and J. Rutten, eds.: *Coalgebraic Methods in Computer Science '98*, vol. 11 of *ENTCS*, pp. 57–72. Elsevier.

Kurz, A. (2000). *A co-variety-theorem for modal logic*. In Zakharyaschev, A., K. Segerberg, M. de Rijke, and H. Wansing, eds.: *Advances in Modal Logic*, vol. 2, pp. 385 – 398.

Kurz, A. (2002). *Logics admitting final semantics*. In Nielsen, M. and U. Engberg, eds.: *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002, Proceedings*, vol. 2303 of *Lecture Notes in Computer Science*, pp. 238–249. Springer-Verlag.

Lambooij, P. (2000). *The YAPI protocol for buffered data transfer*. Techn. Rep. CSI-R9923, Comput. Sci. Inst., Univ. of Nijmegen. Available at URL http://www.cs.kun.nl/csi/reports/info/CSI-R9923.html.

Lamport, L. (1994). *The temporal logic of actions*. ACM Transactions on Programming Languages and Systems, 16(3):872–923.

Lawvere, F. William (1970). *Equality in hyperdoctrines and the comprehension schema as an adjoint functor*. In *Applications of Category Theory*, vol. 17 of *Proceedings of A.M.S. Symposia on Pure Mathematics*, Providence, R.I. American Mathematical Society.

Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs (2000). *JML: notations and tools supporting detailed design in Java*. In *OOPSLA 2000 Companion*, pp. 105–106, Minneapolis, Minnesota. ACM.

Leroy, X., D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon (2001). *The Objective Caml system, release 3.04*. Available at URL http://caml.inria.fr/ocaml/htmlman/.

Liskov, B. (1988). *Data abstraction and hierarchy*. ACM SIGPLAN Notices, 23(5):17–34.

Mac Lane, S. (1997). *Categories for the Working Mathematician*, vol. 5 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, 2nd ed. (1st ed., 1971).

Mandel, L. and M. V. Cengarle (1999). *On the expressive power of OCL*. In *FM'99 - Formal Methods. Proceedings, Volume I*, vol. 1708 of *Lecture Notes in Computer Science*, pp. 854–874. Springer.

Meyer, B. (1992). *Eiffel: The Language*. Prentice Hall.

Meyer, B. (1997). *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA.

Meyer, D. (1999). *A case study in object oriented specification and verification: The MSMIE protocol*. Graduation thesis, Catholic University of Nijmegen.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Milner, R., M. Tofte, and R. Harper (1991). *The Definition of Standard ML*. MIT Press, Cambridge, MA.

Mitchell, J. C. and G. D. Plotkin (1988). *Abstract types have existential types*. ACM Trans. on Programming Languages and Systems, 10(3):470–502.

Moss, Lawrence S. (1999). *Coalgebraic logic*. Annals of Pure and Applied Logic, 96(1–3):277–317.

Mossakowski, T. (2000). *CASL: From semantics to tools*. In Graf, S. and M. Schwartzbach, eds.: *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 1785 of *Lecture Notes in Computer Science*, pp. 93–108. Springer.

Mosses, P. D. (1997). *CoFI: The common framework initiative for algebraic specification and development*. In Bidoit, M. and M. Dauchet, eds.: *TAPSOFT '97: Theory and Practice of Software Development*, vol. 1214 of *Lecture Notes in Computer Science*, pp. 115–140. Springer-Verlag.

Nipkow, T., L. C. Paulson, and M. Wenzel (2002a). *Isabelle's Logics: HOL*. Available via http://isabelle.in.tum.de/.

Nipkow, T., L. Paulson, and M. Wenzel (2002b). *Isabelle/HOL, A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*. Springer-Verlag.

OMG (1997). *Object Constraint Language Specification, version 1.1*. Object Management Group. OMG Document ad/97-08-08.

OMG (2001). *OMG Unified Modeling Language Specification, Version 1.4*. Object Management Group. OMG document formal/01-09-67.

Owre, S., S. Rajan, J. Rushby, N. Shankar, and M. Srivas (1996). *PVS: Combining specification, proof checking, and model checking*. In Alur, R. and T. Henzinger, eds.: *Computer Aided Verification*, vol. 1102 of *Lecture Notes in Computer Science*, pp. 411–414. Springer, Berlin.

Owre, S., J. Rushby, N. Shankar, and F. von Henke (1995). *Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS*. IEEE Trans. on Softw. Eng., 21(2):107–125.

Owre, S. and N. Shankar (1993). *Abstract datatypes in PVS*. Technical Report SRI-CSL-93-9R, Computer Science Laboratory, SRI International, Menlo Park, CA. Extensively revised June 1997; Also available as NASA Contractor Report CR-97-206264.

Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert (1999a). *PVS Language Reference, Version 2.3*. Computer Science Laboratory, SRI International, Menlo Park, CA.

Owre, S., N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert (1999b). *PVS System Guide, Version 2.3*. Computer Science Laboratory, SRI International, Menlo Park, CA.

Paulson, L. C. (2002). *Introduction to Isabelle*. Available via http://isabelle.in.tum.de/.

Phoa, W. (1992). *An introduction to fibrations, topos theory, the effective topos and modest sets*. Research report ECS-LFCS-92-208, Lab. for Foundations of Computer Science, University of Edinburgh.

Pierce, B. and M. Steffen (1997). *Higher-order subtyping*. Theoretical Computer Science, 176(1–2):235–282.

Pierce, B. C. and D. N. Turner (1994). *Simple type-theoretic foundations for object-oriented programming*. Journal of Functional Programming, 4(2):207–247.

Poll, E. (2001). *A coalgebraic semantics of subtyping*. Theoretical informatics and applications, 35(1):61–81.

Poll, E. and J. Zwanenburg (2001). *From algebras and coalgebras to dialgebras*. In Corradini, A., M. Lenisa, and U. Montanari, eds.: *Coalgebraic Methods in Computer Science '01*, vol. 44 of *ENTCS*. Elsevier, Amsterdam.

Reichel, H. (1985). *Behavioural validity of conditional equations in abstract data types.* In *Contributions to General Algebra 3*. Teubner. Proceedings of the Vienna Conference, June 21-24, 1984.

Reichel, H. (1995). *An approach to object semantics based on terminal co–algebras.* Mathematical Structure in Computer Science, 5:129–152.

Reichel, H., ed. (2000). *Workshop on Coalgebraic Methods in Computer Science '00*, vol. 33 of *ENTCS*, Berlin. Elsevier.

Reynolds, John C. (1984). *Polymorphism is not set-theoretic.* In Kahn, G., D. B. MacQueen, and G. Plotkin, eds.: *Proceedings Int. Symp. on the Semantics of Data Types, Sophia-Antipolis, France, 27–29 June 1984*, vol. 173 of *Lecture Notes in Computer Science*, pp. 145–156. Springer-Verlag, Berlin.

Richters, M. and M. Gogolla (1998). *On formalizing the UML object constraint language OCL.* In Ling, T.-W., S. Ram, and M. L. Lee, eds.: *Proc. 17th International Conference on Conceptual Modeling (ER)*, vol. 1507 of *Lecture Notes in Computer Science*, pp. 449–464. Springer-Verlag.

Richters, M. and M. Gogolla (2002). *OCL: Syntax, semantics, and tools.* In Clark, T. and J. Warmer, eds.: *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, vol. 2263 of *Lecture Notes in Computer Science*, pp. 42–68. Springer.

Rößiger, M. (1999). *Languages for coalgebras on datafunctors*. In Jacobs, B. and J. Rutten, eds.: *Coalgebraic Methods in Computer Science*, vol. 19 of *ENTCS*. Elsevier, Amsterdam.

Rößiger, M. (2000a). *Coalgebras and modal logic.* In Reichel, H., ed.: *Coalgebraic Methods in Computer Science '00*, vol. 33 of *ENTCS*. Elsevier Science Publishers.

Rößiger, M. (2000b). *Coalgebras, Clone Theory and Modal Logic.* PhD thesis, Univ. of Dresden, Germany.

Roşu, Grigore (2000). *Hidden Logic.* PhD thesis, University of California at San Diego.

Rothe, J. (2000). *Modal logics for coalgebraic class specification.* Master's thesis, TU Dresden, Germany. Available at URL: http://wwwtcs.inf.tu-dresden.de/∼janr/diplom.ps.gz.

Rothe, J., H. Tews, and B. Jacobs (2001). *The coalgebraic class specification language CCSL.* Journal of Universal Computer Science, 7(2):175–193.

Rutten, J. J. M. M. (2000). *Universal coalgebra: A theory of systems.* Theoretical Computer Science, 249(1):3–80.

Schmidt, D. C. (1990). *gperf: a perfect hash function generator*. In *USENIX C++ Conference*, pp. 87–101, Berkeley, CA, USA. USENIX Association.

Schroeder, M. A. (1997). *Higher order charity*. Master's thesis, The University of Calgary, Department of Computer Science.

Secure Networks (1998). *Custom Attack Simulation Language (CASL), User manual*. Secure Networks.

Stärk, R. F., J. Schmid, and E. Börger (2001). *Java and the Java Virtual Machine: definition, verification, validation*. Springer-Verlag, Berlin.

Stirling, Colin (1992). *Modal and temporal logics*. In Abramsky, S., D. M. Gabbay, and T. Maibaum, eds.: *Handbook of Logic in Computer Science*, vol. 2, chap. Modal and Temporal Logics. Oxford Science Publications.

Stroustrup, B. (1997). *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass.

Tarski, Alfred (1955). *A lattice-theoretic fixpoint theorem and its applications*. Pacific Journal of Mathematics, 5(2):285–309.

Tews, H. (2000a). *A case study in coalgebraic specification: Memory management in the* FIASCO *microkernel*. Technical Report TPG2/1/2000, SFB 358. Available at URL http://wwwtcs.inf.tu-dresden.de/~tews/vfiasco/.

Tews, H. (2000b). *Coalgebras for binary methods*. In Reichel, H., ed.: *Coalgebraic Methods in Computer Science '00*, vol. 33 of *ENTCS*. Elsevier, Amsterdam.

Tews, H. (2001). *Coalgebras for binary methods: Properties of bisimulations and invariants*. Theoretical informatics and applications, 35(1):83–111.

Tews, H. (2002a). *The Coalgebraic Class Specification Language CCSL (Reference manual)*. Available via URL wwwtcs.inf.tu-dresden.de/~tews/ccsl/.

Tews, H. (2002b). *Greatest bisimulations for binary methods*. In Moss, L., ed.: *Coalgebraic Methods in Computer Science '02*, vol. 65.1 of *ENTCS*. Elsevier, Amsterdam.

U. Schöning (1997). *Theoretische Informatik kurz gefasst*. Akademischer Verlag, 3. Aufl.

Vigna, G., S. Eckmann, and R. Kemmerer (2000). *Attack languages*. In *Proceedings of the IEEE Information Survivability Workshop*, Boston, MA. available via http://www.cs.ucsb.edu/~vigna/.

Warmer, J. and A. Kleppe (1999). *The Object Constraint Language: Precise Modeling with UML.* Addison Wesley, Reading, Mass.

Warmer, J., A. Kleppe, T. Clark, A. Ivner, J. Högström, M. Gogolla, M. Richters, H. Hussmann, S. Zschaler, S. Johnston, D. S. Frankel, and C. Bock (2001). *Object constraint language 2.0, Submission to the OMG*. Technical report, University of Bremem. OMG Document ad/2001-08-01.

Wenzel, M. (2002). *The Isabelle/Isar Reference Manual*. Available via http://isabelle.in.tum.de/.

Wirsing, M. (1990). *Algebraic specification*. In Leeuwen, J. van, ed.: *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics, chap. 13, pp. 673–788. Elsevier/MIT Press.

Notation Index

The notation index is split into six parts: (1) the index for derivation rules, (2) the index of logical entailments, (3) the index of fibrations, (4) the index of symbols based on the Greek alphabet, (5) the index of symbols based on the Latin alphabet (on page 300), and (6) the index of other symbols (on page 302).

Rule Index

Entailment Index

Fibration Index

Subject Index

Entries in `UPPERCASE TYPEWRITER` are CCSL keywords. Entries in *slanted* are meta symbols. For both kinds of entries page numbers smaller than 281 refer into Chapter 4, larger page numbers refer into Appendix B with the full CCSL grammar.